

Designing, Manufacturing, and Predicting Deformation of a Formable Crust Matrix

A Thesis
Presented to
The Academic Faculty

By

Austina N. Nguyen

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science in Mechanical Engineering

Georgia Institute of Technology

July 2004

Copyright © 2004 by Austina N. Nguyen

Designing, Manufacturing, and Predicting Deformation of a Formable Crust Matrix

Approved:

Dr. David Rosen
Chairman of the Committee
Associate Professor
Mechanical Engineering

Dr. Mark Allen
Professor
Electrical & Computer Engineering

Dr. Imme Ebert-Uphoff
Assistant Professor
Mechanical Engineering

Date Approved: 16 June 2004

Suffering is but another name for the teaching of experience, which is the parent of
instruction and the schoolmaster of life

-- [Horace](#)

And for this thesis, I suffered . . . and learned.

ACKNOWLEDGEMENTS

My Parents

Because of the incomplete translation of English words to Vietnamese, it took them a while to realize that I am a graduate researcher and not a paper filer, a mechanical design engineer and not a car mechanic or fashion designer. But no matter what I strived to do, they have been there for me.

Dr. David Rosen

A man who would not take NO for an answer. He pushed and pushed me to complete every task even when it seemed impossible, reminding me, “It is not like putting a man on the moon.” Then, he pushed and pushed me to get the heck outta his department! Because of him, I am a mechanical engineer.

Dr. Imme Ebert-Uphoff and Dr. Mark Allen

I’d like to thank my other committee members for taking the time to read this thesis. I know that from their inputs I can make this thesis a document to look back upon with pride: “Yes my grandchild, I wrote that”.

The Lab-mates in MARC 251

In the room where the lights remain on late into the night and into the early part of the morning, there are always companies, words of encouragement, and people full of

thesis experiences. I'd like to thank personally Sundiata Jangha and Benay Sager for sharing with me their experiences at Georgia Tech.

Loren Ybarrondo

During the first half of my thesis work, I was struggling. He was there entertaining me, keeping up my spirit and staying up late at night to help me develop the next set of equations. Because of him, I gained one of the greatest rewards from being a graduate student: learning endurance. Loren, you are definitely a true friend.

Kyle Mitchell

The soon to be famous writer who was kind enough to lend me his creativity by editing my thesis, I owe him much thanks.

Jason Lawrence

The one whom I owe the most debt for making this thesis possible: Jason. He was there from the beginning to the end. He was there teaching me how to code in MATLAB. He was there fine-tuning my knowledge about spherical coordinates. He was there to the very end, reading over my thesis one page at a time. Because of him, I have made it this far. Thank-you so much.

This research was financially supported by the Rapid Prototyping Manufacturing Institute (RPMI) at Georgia Institute of Technology and the National Science Foundation: grant number IIS-0121663.

TABLE OF CONTENTS

ACKNOWLEDGEMENT	iv
TABLE OF CONTENTS	vi
LIST OF TABLES	xi
LIST OF FIGURES	xiii
SUMMARY	xix

CHAPTER 1

INTRODUCTION TO DIGITAL CLAY	1
1.1 Digital Clay Context	1
1.2 Motivation for Studying Digital Clay	3
1.3 The Benefits of Digital Clay to Society	4
1.4 Digital Clay Team	4
1.5 Digital Clay Architecture	6
1.6 Problem Statement	8
1.7 Key Question	9
1.8 Goals	10
1.9 Development Questions	11
1.10 Development Hypothesis	12
1.11 Deliverables: Show Me The Money!	14
1.12 Introduction to The Rest of The Thesis	15

CHAPTER 2

LITERATURE REVIEW	16
2.1 “Virtual Clay” Ideas	16
2.1.1 CyberGrasp and RM-II Hand Master	16
2.1.2 The PHANTOM	18
2.1.3 FreeForm	19

2.1.4 Feelex	20
2.2 Elastic Deformation Products	21
2.2.1 Compliant Mechanism	22
2.2.2 Flexural-Based Gripper	23
2.3 Ending Comment	24
 CHAPTER 3	
THE SKETCHBOOK OF MATRIX DESIGNS	25
3.1 Requirement List	25
3.2 Check (Clarifying The Task)	27
3.3 Abstracting to Identify the Essential Problem	28
3.3.1 Abstraction and Problem Formation and Systematic Broadening	28
3.4 Function Structure	29
3.5 The Manufacturing Technique	30
3.5.1 MEMS	30
3.5.1.1 Thermal Press Molding	31
3.5.1.2 Injection Molding	31
3.5.1.3 PDMS Cast Molding	31
3.5.1.4 Lamination	31
3.5.2 LCVD	32
3.5.3 Other Techniques	33
3.5.4 SLA	33
3.6 Design and Manufacturing of Designs	35
3.6.1 The Flexible Corners	35
3.6.2 Deformable Cubes	36
3.6.3 Compliant Hinges	37
3.6.4 Deformable Crust Design	38
3.6.4.1 Unit Cell for Crust	38
3.6.4.1.1 Eight-Sided Unit Cell	39
3.6.4.1.2 Spherical Joint Unit Cell	40
3.6.4.1.3 Linear Triangles	41

3.6.5 Selection Process of the Unit Cells for the Crust Matrix	42
3.6.6 Matrix Selection	44
3.6.7 Modification of Selected Unit Cell	46
3.6.8 Grid Matrix	47
3.6.9 Hexagon Matrix	48
3.7 Crust Matrix MEMS Style	50
3.8 Ending Remarks	53

CHAPTER 4

IT IS THE PRINCIPLES BEHIND THE MATH 55

PART 1 METHOD 1:	64
ABSTRACT FORMABLE CRUST MODEL	
4.1 Referencing Notation	68
4.2 The Principles Underlying the Formable Crust Models	69
4.3 Constraints From User Inputs (Method 1: “User Inputs” Block)	69
4.4 Line Interpolation (Method 1: “Interpolate” block)	70
4.5 Constraints (Method 1: “Iteration” Block)	71
4.5.1 Length Constraints	71
4.5.2 Coordinate Constraints	71
4.6 Calculating Energy (Method 1: “Iteration” Block)	72
4.6.1 Finding the Stiffness Value	73
4.6.1.1 The Experimental Set-up	74
4.6.1.2 Joint A (Larger Joint) Design and Results	74
4.6.1.3 Joint B (Smaller Joint) Design and Results	78
4.6.1.4 Effect of k	80
4.6.1.5 Stiffness Ending Remarks	83
4.6.2 Calculating the Angles from the Position Coordinates	84
4.7 Iteration Process (Method 1: “Iteration” Block)	84
4.7.1 Determination of a Direction using the Hessian Matrix	85
4.7.2 Line-Search Procedure	86
4.8 Ending Comments for Method 1	86

PART 2 METHOD TWO:	87
ACTUAL MANUFACTURABLE CRUST MODEL	
4.9 Spherical Coordinates (Method 2: Interpolation and Initial Guess)	92
4.10 Initial Guess (Method 2: Interpolate and Initial Guess)	95
4.11 Duplications (Method 2: “Create Relationship” Block)	98
4.12 From Spherical to Cartesian (Method 2: Inside Iteration Box)	100
4.13 Calculating the Joint Angles	
(Method 2: Joint Angles Calculation)	105
4.13.1 Joint Angle Pseudo-code for One Unit Cell	106
4.13.2 Joint Angle Calculation for an Array	112
4.14 Potential Energy (Method 2: Inside Iteration Box)	114
4.15 Ending Comments for Method 2	115
4.16 Forward and Inverse Statics: Overall Statics of Method 2	116
4.16.1 Inverse Statics	118
4.16.1.1 Inverse Statics Equations	119
4.16.2 Forward Statics	120
4.16.2.1 Forward Statics Algorithm	122
4.17 Unknowns, Equations, and Degrees-of-Freedom	125
4.17.1 Method 1	127
4.17.2 Method 2	131
4.17.3 Ending Remarks for Section	137
4.18 Ending Remarks for Chapter	139
4.18.1 Benefits	139
 CHAPTER 5	 141
EXPERIMENTS AND RESULTS	
5.1 Compare and Contrast	142
5.2 Line Test 1	145
5.3 Line Test 2	150
5.4 Line Test 3	154
5.5 Line Test 4: Piggybacking Style	160

5.5.1 Piggybacking Style Example 1	161
5.5.2 Piggybacking Style Example 2	164
5.6 Method 1 Matrix Elapse Time Study	168
5.7 Method 1 Matrix Accuracy Prediction	171
5.7.1 Matrix Plane Test 1	171
5.7.2 Matrix Surface Test 2	172
5.8 Mimicking the Car-hood Models	173
5.9 Degeneracy	175
5.10 Uniqueness	176
5.11 Geometric Non-linearity	181
5.12 Ending Remarks	182
5.13 Comparing Two Theses	182
 CHAPTER 6	
LAST(ING) COMMENTS	185
6.1 Concluding Comments	185
6.2 Future works	188
6.3 Benefits and Values	191
 APPENDIXES	
APPENDIX A FINDING STIFFNESS VALUE	194
APPENDIX B JOINT ANGLES DEFORMATION	210
APPENDIX C MATLAB FOR METHOD 1	218
APPENDIX D MATLAB FOR METHOD 2: FIX FACE	260
APPENDIX E MATLAB FOR METHOD 2: FREE FACE	294
 BILIOGRAPHY	322

LIST OF TABLES

Table 1.1: Digital Clay Team Members	5
Table 3.1: Requirement List	26
Table 3.2: Requirement List (continued)	27
Table 3.3: Manufacturing Attribute	42
Table 3.4: Scalable Attribute	42
Table 3.5: Deformation Attribute	43
Table 3.6: Rank of the Criteria	43
Table 3.7: Design Section Table	43
Table 4.1: Finding Unit Vector \bar{a}_1 and \bar{a}_3 from Bosscher's Master's Thesis	107
Table 4.2: Finding the Joint Angles θ 's From Bosscher's Master Thesis	111
Table 5.1: Computer Configuration	142
Table 5.2: Table 5.2: Kinematics Comparison Test 1	146
Table 5.3: Time Comparison for Line Test 1	147
Table 5.4: Iteration and Energy Comparison for Line Test 1	148
Table 5.5: Z-values Comparison	149
Table 5.6: Kinematics Comparison Test 2	151
Table 5.7: Time Comparison for Line Test 2	151
Table 5.8: Iteration and Energy Comparison for Line Test 2	152
Table 5.9: Z-values Comparison	153
Table 5.10: Kinematics Comparison Test 3	155
Table 5.11: Time Comparison for Line Test 3	155
Table 5.12: Iteration and Energy Comparison for Line Test 3	156
Table 5.13: Z-values Comparison	157
Table 5.14: Joint Results for Method 1	159

Table 5.15: Joint Results for Method 2 Fix	159
Table 5.16: Joint Results for Method 2 Free	159
Table 5.17: Time Comparison for Piggyback Example 1	162
Table 5.18: Iteration and Energy Comparison for Piggyback Example 1	162
Table 5.19: Z-values Comparison	163
Table 5.20: Time Comparison for Piggyback Example 2	165
Table 5.21: Iteration and Energy Comparison for Piggyback Example 2	165
Table 5.22: Z- values Comparison	166
Table A.1: Summary of Stiffness Values for Large Joint	204
Table A.2: Summary of Stiffness Values for Small Joint	209
Table B.1: Joint Results For Cell 1 and 2. Method 1	212
Table B.2: Joint Results For Cell 3 and 4. Method 1	212
Table B.3: Joint Results For Cell 5 and 6. Method 1	213
Table B.4: Joint Results For Cell 7. Method 1	213
Table B.5: Joint Results For Cell 1 and 2. Method 2: Fix	214
Table B.6: Joint Results For Cell 3 and 4. Method 2: Fix	215
Table B.7: Joint Results For Cell 5 and 6. Method 2: Fix	215
Table B.8: Joint Results For Cell 7. Method 2: Fix	215
Table B.9: Joint Results For Cell 1 and 2. Method 2: Free	216
Table B.10: Joint Results For Cell 3 and 4. Method 2: Free	217
Table B.11: Joint Results For Cell 5 and 6. Method 2: Free	217
Table B.12: Joint Results For Cell 7. Method 2: Free	217

LIST OF FIGURES

Figure 1.1: Schematic of Digital Clay in Use	1
Figure 1.2: The Overall Goals of Digital Clay	2
Figure 1.3: Flow Chart of Task	5
Figure 1.4: Control and Interface Subsystems	7
Figure 2.1: CyberGrasp (Left) and RM-II Hand Master (Right)	17
Figure 2.2: Face deformation (www.siggraph.org)	18
Figure 2.3: SensAble Technology PhanTom	19
Figure 2.4: FreeForm Modeling for Manufacturing	20
Figure 2.5: Feelex Version 1 and 2	21
Figure 2.6: Compliant Crimping (Left). Compliant Gripper (Right)	22
Figure 2.7: Flexural-Based Gripper Design (Left). Manufactured (Right)	23
Figure 3.1: Overall Function Structure	29
Figure 3.2: Function Structure Using Generally Valid Functions	30
Figure 3.3: An Additive Fabrication Process – Stereolithography (Jacobs, 1992)	34
Figure 3.4: The Flexible Corners	35
Figure 3.5: Deformable Cubes	36
Figure 3.6: Compliant Hinges	37
Figure 3.7: Deformable Crust Concepts	38
Figure 3.8: Twelve-sided Deformable Crust	39
Figure 3.9: Spherical Joint Unit Cell	40
Figure 3.10: Linear Triangles	41
Figure 3.11: Grid Matrix Unit Cell	47
Figure 3.12: Grid Matrices	47
Figure 3.13: Hexagon Unit Cell	48
Figure 3.14: Hexagon Matrices	49

Figure 3:15: Bellows Bubble Actuators Concept 1	51
Figure 3:16: Bellows Bubble Actuators Concept 2	51
Figure 3.17: One Unit Cell	52
Figure 3.18: Enclosing Membrane	52
Figure 4.1:Crust Matrix	55
Figure 4.2: Actual Crust Matrix Deforming	56
Figure 4.3: Low Degree-of-Freedom Car Hood Model	57
Figure 4.4: Deformation of Car Hood Model	58
Figure 4.5: Morphing of the Car Hoods	58
Figure 4.6: Car Hood Frame	59
Figure 4.7: 2D Example of Input	60
Figure 4.8: Basic Structure of Both Methods	61
Figure 4.9: Method 1-- Abstract Model	62
Figure 4.10: Springs for One Unit Cell	62
Figure 4.11: Flow Chart of Method 1	64
Figure 4.12: Example of matrix initial conditions	65
Figure 4.13: the Angles Between Two Center-points	66
Figure 4.14: Counting Convention	68
Figure 4.15: Inputting Z-heights	69
Figure 4.16: Line interpolation	70
Figure 4.17: The Two Different Joint Designs	73
Figure 4.18: Joint A (Left) and Joint B (Right)	73
Figure 4.19: Experimental set-up for Finding Stiffness	74
Figure 4.20: Set-up for Joint A	75
Figure 4.21: Finding Stiffness for the Joint A—the Larger Joint.	77
Figure 4.22: Set-up for Joint B	78
Figure 4.23: Finding Stiffness for Joint B (smaller Joint)	79
Figure 4.24: 4-by-5 Method 1 K Test. $K=1$ (1 st image). $K=100$ (3 rd). $K=1000$ (4 th)	81
Figure 4.25: $K_1=100$; $K_2=500$ (LT). $K_1=500$; $K_2=500$ (M). $K_1=100$; $K_2=1000$ (RT)	83
Figure 4.26: Flow Chart of Method 2	88

Figure 4.27: The \vec{n} and \vec{v} -vectors	93
Figure 4.28: Spherical Coordinates	93
Figure 4.29: Example of Spherical Coordinates	94
Figure 4.30: Interpolation From Z-input	96
Figure 4.31: \vec{n} and \vec{v} -vectors for One Unit Cell	98
Figure 4.32: Linking unit cells	99
Figure 4.33: Rotation about J-axis	101
Figure 4.34: Rotation about K-axis	102
Figure 4.35: Two Rotations	103
Figure 4.36: Unit Cell	106
Figure 4.37: Detail of Dashed Circle of Unit Cell	108
Figure 4.38: Second Detail	110
Figure 4.39: Array of Unit cells with Center points	112
Figure 4.40: Arrangement of \vec{a} and \vec{a}_{mid} vectors	113
Figure 4.41: Forward and Inverse Statics	116
Figure 4.42: Calculates the Position	117
Figure 4.43: Inverse Statics Diagram	119
Figure 4.44: Calculating the Edge Vertices	120
Figure 4.45: Linking Unit Cells	122
Figure 4.46: Roll-Pitch-Yaw	126
Figure 4.47: Numbers of Unknowns and Constraints for a 1-by-7 Matrix	127
Figure 4.48: Method 1 2D Example for DoF	129
Figure 4.49: 4-by-5 Matrix of Constraints	129
Figure 4.50: 2-by-2 DoF Example	130
Figure 4.51: Method 1: 4-by-5 DoF Example	131
Figure 4.52: Method 2: DoF Counting	134
Figure 4.53: Method 2: 2-by-2 DoF Example	135
Figure 4.54: Method 2. 4-by-5 DoF Example	136

Figure 5.1: Fixed Face: Line of Cells (Left) and Detail of Fixed Face (Right)	143
Figure 5.2: Free Face: Line of Cells (Left) and Detail of Free Face (Right)	143
Figure 5.3: Line Test 1 Method 1: Abstract Model	145
Figure 5.4: Line Test 1 Method 2: With Fixed Face	145
Figure 5.5: Line Test 1 Method 2: Without Fixed Face	145
Figure 5.6: Z-Values Results for Line Test 1	149
Figure 5.7: Line Test 2 Method 1--Abstract Model	150
Figure 5.8: Line Test 2 Method 2-- With Fixed Face	150
Figure 5.9: Line Test 2 Method 2-- Without Fixed Face	150
Figure 5.10: Z-Values Results for Line Test 2	152
Figure 5.11: Line Test 2 Method 1-- Abstract Model	154
Figure 5.12: Line Test 2 Method 2--With Fixed Face	154
Figure 5.13: Line Test 2 Method 2--Without Fixed Face	154
Figure 5.14: Z-Values Results for Line Test 3	157
Figure 5.15: Graphic Reminder of Previous Results for Method 1 and 2	161
Figure 5.16: Piggybacked Example 1 Result	161
Figure 5.17: Z-Values for Piggyback Example 1	163
Figure 5.18: Method 1 Inputs (Left) and Line Interpolation (Right)	164
Figure 5.19: Method 1 Results	164
Figure 5.20: Piggyback Results	164
Figure 5.21: Z-Values for Piggyback Example 2	166
Figure 5.22: 3-by-3 Matrix Input at [2,2] (Left). Results (Right)	168
Figure 5.23: 4-by-4 Matrix Input at [3,3] (Left). Results (Right)	168
Figure 5.24: 5-by-5 Matrix Input at [3,3] (Left). Results (Right)	169
Figure 5.25: Computational Time	169
Figure 5.26: Iteration Rate	170
Figure 5.27: Plane Test Inputs (Left). Results (Right)	171
Figure 5.28: Surface Test Inputs (Left). Results (Right)	172
Figure 5.29: Morphing Car-hood Models	173
Figure 5.30: Ferrari Attempt	173

Figure 5.31: Corvette Results	174
Figure 5.32: Non-Existing Springs	175
Figure 5.33: Non-deformed state (Left) Shear Deformation (Right)	176
Figure 5.34: The Bowing Effects	177
Figure 5.35: Set 1. Flat Plane Starting Position (LT). Input (M). Results (RT)	178
Figure 5.36: Set 2. Flat Plane Starting Position (LT). Input (M). Results (RT)	179
Figure 5.37: Set 3 Starting Position with Inputs	179
Figure 5.38: Set 3. Starting Position (LT). Input (M). Results (RT)	180
Figure 5.39: Bosscher's Abstract model	183
Figure A.1: Results Large Joint Run 1	195
Figure A.2: Stiffness Value for Large Joint. Run 1	195
Figure A.3: Results Large Joint Run 2	196
Figure A.4: Stiffness Value for Large Joint. Run 2	196
Figure A.5: Results Large Joint Run 3	197
Figure A.6: Stiffness Value for Large Joint. Run 3	197
Figure A.7: Results Large Joint Run 4	198
Figure A.8: Stiffness Value for Large Joint. Run 4	198
Figure A.9: Results Large Joint Run 5	199
Figure A.10: Stiffness Value for Large Joint. Run 5	199
Figure A.11: Results Large Joint Run 6	200
Figure A.12: Stiffness Value for Large Joint. Run 6	200
Figure A.13: Results Large Joint Run 7	201
Figure A.14: Stiffness Value for Large Joint. Run 7	201
Figure A.15: Results Large Joint Run 8	202
Figure A.16: Stiffness Value for Large Joint. Run 8	202
Figure A.17: Results Large Joint Run 9	203
Figure A.18: Stiffness Value for Large Joint. Run 9	203
Figure A.19: Results Small Joint Run 1	205
Figure A.20: Stiffness Value for Small Joint. Run 1	205

Figure A.21: Results Small Joint Run 2	206
Figure A.22: Stiffness Value for Small Joint. Run 2	206
Figure A.23: Results Small Joint Run 3	207
Figure A.24: Stiffness Value for Small Joint. Run 3	207
Figure A.25: Results Small Joint Run 4	208
Figure A.26: Stiffness Value for Small Joint. Run 4	208
Figure B.1: Results from Method 1	211
Figure B.2: Results from Method 2: Fix Face	214
Figure B.3: Results from Method 2 Free Face	216

SUMMARY

Digital Clay represents a new type of 3-D human-computer interface device that enables tactile and haptic interactions. The Digital Clay kinematics structure is computer controlled and can be commanded to acquire a wide variety of desired shapes (shape display), or be deformed by the user in a manner similar to that of real clay (shape editing). The design of the structure went through various modifications where we finally settled on a crust matrix of spherical joint unit cells. After designing the kinematics structure, the next step is predicting the deformation of the crust matrix based upon a handful of inputs. One possible solution for predicting the shape outcome is considering minimizing the potential energy of the system. In this thesis two methods will be introduced. The first method will be an abstract model of the crust where the energy is calculated from a simplified model with one type of angular springs. The second method is the actual manufacturable crust model with two types of angular springs. From the implementation of these two methods, the output will be center-points of the unit cells. From the center-points, one can also calculate the joint angles within each unit cell.

CHAPTER 1

INTRODUCTION TO DIGITAL CLAY

Digital Clay is the next stage of CAD modeling. The main focus of this thesis will be the manufacturability and kinematics of the Digital Clay structure that will help this advancement.

1.1 Digital Clay Context

In recent years, the communication of form and complex data has been greatly enhanced by visualization technologies. However these visualization technologies are based on planar images. With the advancement of computational power, it is now possible to consider real-time, tactile 3-D physical communication to overcome the inherent limitations of planar images. A team at the Georgia Institute of Technology is pursuing a novel type of human-computer interaction called Digital Clay. Figure 1.1 shows a schematic illustration of Digital Clay being used for shape editing.

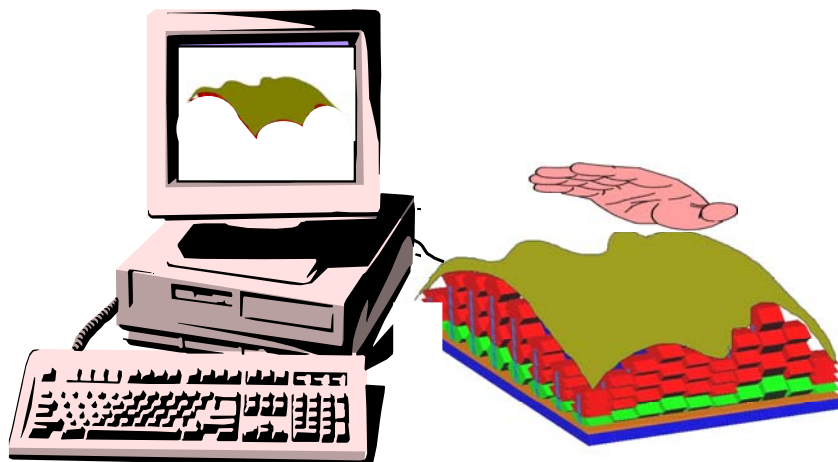


Figure 1.1: Schematic of Digital Clay in Use.

The objective of the Digital Clay, NSF-sponsored research team is to develop an interactive technique that combines haptic sensation with computer algorithms to achieve two key goals. The first goal is to design a deformable, spatially-continuous surface with sensors that store its shape in a computer as the user deforms the surface. In addition, the surface will be actuated so that users can input shape data into the computer and the Digital Clay will deform itself into the desired shape. The shape data can be sent and received electronically anywhere within the world through the use of the Internet. This will allow other users to not only see an image of the surface but alter its shape as well. The second goal is to provide this visual and haptic sensation simultaneously through the use of a single device that does not obligate an individual to wear any extra apparatus (gloves, virtual headgear, etc.). Figure 1.2 shows a schematic of how the Digital Clay product can inter-connect people.

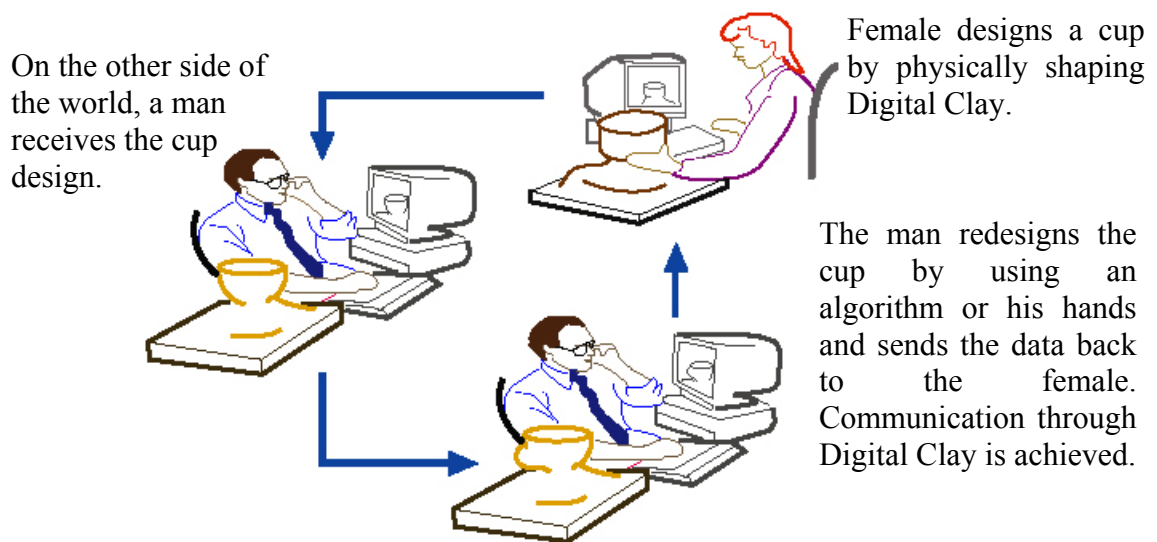


Figure 1.2: The Overall Goals of Digital Clay

This thesis will discuss the development of one feature of the Digital Clay device: the deformable kinematics structure that takes on shape and displays the shape. Later this thesis will discuss the technical issues and mathematics behind generating arbitrary shapes in the clay.

1.2 Motivation for Studying Digital Clay

Currently we are living in a world that is progressing toward global communication. For example in the production of automobiles, the body of the car may be designed in Germany, the engine may be designed in the U.S., and the whole car may be assembled in Mexico. How do all of these people communicate their ideas, development, and progression? In the past, we have relied on mail, telephones, and fax machines. But with the advent of the world-wide-web, we have the power to communicate within minutes. However, the Internet only allows us to communicate flat objects like pictures and text, not the real thing. So we revert back to mail. Hey, wait a minute -- what happened to progress? Digital Clay will be the next innovative tool in this world of global communication that will fill up the “progression” gap. This technology will allow designers, engineers, artists, doctors, and lawyers around the world to interact on a visual and somewhat physical aspect.

1.3 The Benefits of Digital Clay to Society

Digital Clay has several potential applications for society. These include:

- 1) Art: Displaying shape morphing
- 2) Medical diagnostics: Studying the shape of cancers
- 3) Bioengineering device design: Fitting artificial limbs for amputee
- 4) Reconfigurable displays: Demonstrating motion as in a wheel on a car
- 5) Products: Designers sharing concepts
- 6) Mechanical computer-aided design: Two gears turning
- 7) Education: Distance collaboration for product development:
- 8) Visually impaired persons: Communication for the blind
- 9) Lawyers: Re-enactment of scenes

Some applications require the user to directly shape the surface, while others only display the shape. For example, for medical diagnosis sometimes it is only necessary to support shape and display stiffness so that the Digital Clay can “feel” like an organ or a type of tissue. In all cases, the Digital Clay device will advance our present knowledge of how we design, communicate, and collaborate.

1.4 Digital Clay Team

The Digital Clay project was initiated in the beginning weeks of July 2001 when NSF (National Science Foundation) financially approved Georgia Institute of Technology to develop a realistic Digital Clay product. The following Georgia Institute of

Technology faculty members and students were given with this task. The names are shown in Table 1.1.

Table 1.1: Digital Clay Team Members

Professor / Student	Focal Points Concerning Digital Clay Project
Mark Allen / Guang Bai	MEMS (Micro-Electro Mechanical Systems)
Wayne Book / HaiHong Zhou	Project Manager/Controls
Ari Glezer / Dan Short	Hydraulics/Fluids
David Rosen / Austina Nguyen	Manufacturing
Jarek Rossignac / Byoung-Moon Kim	Computer Modeling Interface
Imme Ebert-Uphoff / Paul Bosscher	Kinematics / Structure Analysis

Below is a flow chart that describes the task of each department. The dashed blocks describe the focus for this thesis.

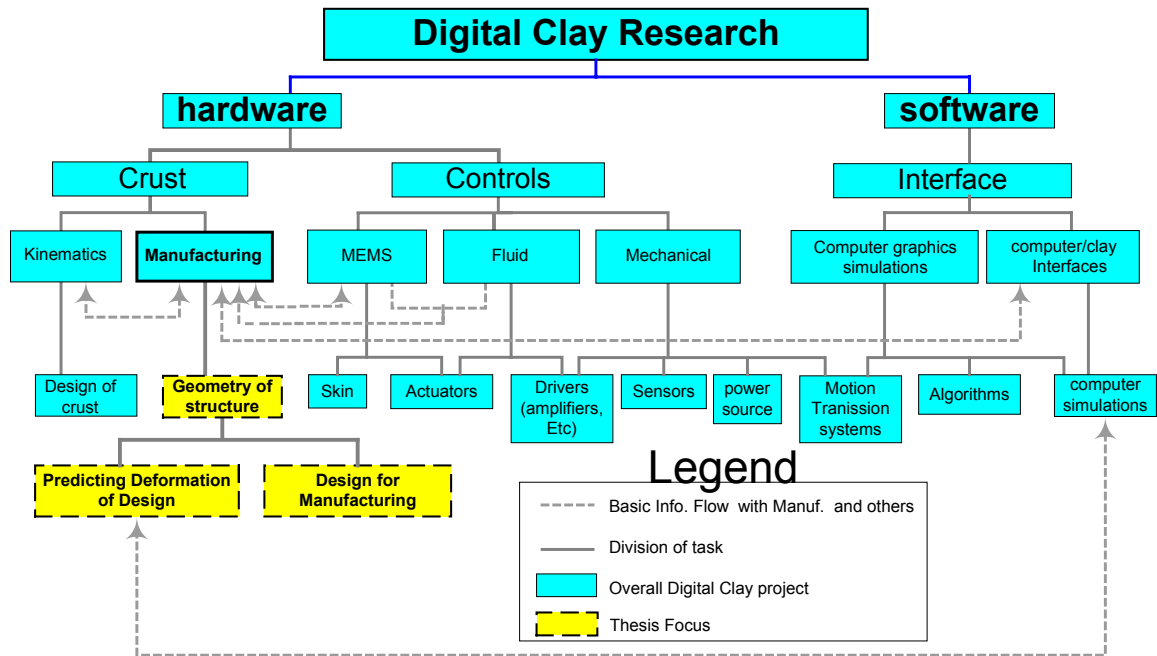


Figure 1.3: Flow Chart of Task

As seen in Figure 1.3, the main focus for this thesis is describing the design of the kinematics structure for quick manufacturing using stereolithography technology and predicting the deformation of the kinematics structure design. For a more descriptive schematic about the interaction within the Digital Clay architecture, refer to the next section. After designing the kinematics structure that will deform, we will need a program that will predict the deformation. We will create this program to test the deformation capability of the kinematics structure that we designed. At the same time, we will communicate with the interface group for feedback and improving the interfacing of the program for human usage.

1.5 Digital Clay Architecture

Digital Clay will be a physical volume bounded by a deformable kinematics structure that acts as the haptic interface. This kinematics structure is connected to a computer that can either recreate the surface topography of the shape inputted as a CAD file or modify the kinematics structure and the volume underneath the surface from a preexisting file. To display any acquired shape either by human manipulation or CAD file, the kinematics structure is controlled by an array of interconnected fluidic-driven actuators. Each actuator is a fluidically inflatable cell that is connected to two common pressurized reservoirs (within a base) through a dedicated two-way miniature valve. The valves and pressure sensors will be part of the Digital Clay base that controls fluid flow to and from the inflatable cells. For measuring deformations and/or displacements of the cells, an additional array of sensors may be incorporated into the base. The Digital Clay

directly indicate to the control unit to inflate or deflate the cells. In either case, the top level of control will first interpret the user's gestures to determine his/her intent. A mathematical model of the clay's behavior will be used to compute commands and parameter values that can drive the clay according to the user's actions. These values are then sent to the lower level clay controller for communication to the actuators. The additional information flow for this mode is shown in bold arrows.

The Digital Clay architecture is a complex maze of interacting subsystems. It is beyond the scope of this thesis to describe every subsystem. The main focus of this thesis is the deformable kinematics structure, which will be discussed in further detail in a later section. The next few sections will explain about the purpose, goals, and focus of the Digital Clay research.

1.6 Problem Statement

The Digital Clay structure is a kinematics structure that deforms to display various shapes. From this initial design idea, there are three problems that arise. The first problem deals with design, the second deals with manufacturing the structure, and the third problem deals with calculating its deformation. Below are the problem statements.

- 1) The kinematics structure needs to be designed such that it can deform into various shapes based upon a set number of inputs.

- 2) A manufacturing process is needed for building the kinematics structure to deform into various shapes without breaking within an aging period of the material used.
- 3) An algorithm is needed to calculate the deformation of the kinematics structure based upon a series of given constraints.
 - a. The algorithm needs be universal enough to consider different types of materials that can be used to manufacture the kinematics structure.
 - b. The algorithm must be computationally efficient and have a fast rate of convergence.

1.7 Key Question

Although the Digital Clay could be built using any number of manufacturing processes, our prototypes were built using stereolithography. Since a production manufacturing process has not been selected, a general focus of study is the deformation of the kinematics structure based upon various materials to help select the material and predict the outcome of the kinematics structure due to applied force. Therefore the main key question is:

What is this kinematics structure and how can the deformation of the kinematics structure be predicted based upon the materials being used and constraints being applied?

1.8 Goals

Based upon the key question, the main goal of this thesis is to design a kinematics structure concept such that it will behave correctly and it is manufacturable. Then an algorithm must be developed that could predict the deformation of the kinematics structure based upon the material properties, system constraints, and user inputs. The results will help the MEMS and the Rapid prototyping departments choose a manufacturing material and help the controls department to predict the deformation.

The goal is broken up into several tasks:

- 1) Designing and manufacturing a deformable kinematics structure using rapid prototyping technology with consideration in scalability, shape generation capability, and longevity to understand how the kinematics structure should deform in real life.
- 2) Expanding the existing joint angle calculation equations for one unit cell of the kinematics structure to calculate all the deformation angles of the whole skin.
- 3) Incorporating the joint stiffness for each unit cell of the kinematics structure from the mechanical properties of Stereolithography material to study the “inverse static” deformation of the structure.
- 4) Developing a “forward statics” algorithm (forward kinematics equations with mechanical properties embedded into the equations) for completing the circle of inverse and forward equations to predict the deformation of the deformable kinematics structure.

1.9 Development Questions

Based on the aforementioned goals, there are two specific areas this thesis will focus upon. One is the design of the kinematics structure for manufacturing. The second is the development of a method for predicting deformation. The development questions are divided into two groups: the design and the deformation method.

1) Design:

- a. What design features should the kinematics structure have to allow the greatest deformation without breaking?
- b. What building process should be considered for designing the kinematics structure?
- c. What size should be chosen such that the kinematics structure yields optimal deformation and is feasible to build?

2) Deformation Method

- a. What is the smallest amount of information needed from the user to determine the deformation of the kinematics structure?
- b. How should the computer algorithm use the inputs to determine the optimal deformation state of the kinematics structure?
- c. What is the best process for guiding the results toward a global solution?
- d. Since this is a static analysis, what properties of the kinematics structure should be known to help determine the deformation of the kinematics structure based upon the manufacturing aspect?
- e. How would the method be implemented?

1.10 Approaches to Answer Development Questions

This section will discuss how the above questions will be answered through a these proposed approaches. Similar to the previous section, this section is divided into two sections, the design and deformation methods.

Design:

- 1) Design a feature that would cause the whole kinematics structure to deform. One possible design focus is the unit cell that makes up the structure, where the unit cell's range of motion would be maximized.
- 2) Consider a manufacturing process that would produce high turn around results. Presently, the kinematics structure is used for studying of deformation and shape formation. We need something that would give quick results with high accuracy. One possible building process is using a rapid prototyping technique called stereolithography where the results can occur within 24 hours.
- 3) Create a kinematics structure that is as large as possible. Since rapid prototyping is suggested, then the size of the kinematics structure should be as large as the platform of the stereolithography machine allowed. For the case of the unit cells that makes up the kinematics structure, the size of the unit cells should be as small as possible. As the size of the cells decreases, the resolution increases which in turns increases the shape formation. A possible size to aim for is 18mm, which is the average width of one fingertip.

Deformation method:

- 1) Require the smallest amount of information possible. The given inputs should create a series of equations for the method. In turn the user is only expected to input only a handful of information to produce the desired outcome. With both types of inputs, this can be an under-constrained problem. Any additional constraints will improve the calculation in either speed or accuracy. To calculate the deformation when there are not enough equations from the given inputs to solve for the number of unknowns, a numerical iterative process can be applied. The numerical method would search for the minimum point similar to Newton Raphson method.
- 2) Improve the initial guess to increase the speed of convergence of the iterative method.
- 3) Determine the material property that is most likely the one that would affect the motion of the unit cells. Since these ranges of motion in these unit cells determine the deformation of the kinematics structure, the most reasonable material property would be the elastic modulus or the stiffness factor.
- 4) Equate the number of equations to the number of unknowns by applying the system of equations method for deriving the answers quicker than the iteration process.

- 5) If the numbers of equations and unknowns are equal, then material properties or material side effects (unit cell stiffness) are not needed to determine the deformation.

1.11 Deliverables

Below are brief descriptions of what this thesis will deliver. The two main deliverables are the manufactured kinematics structure and the deformation algorithm. Since it is difficult to determine the best way to predict the deformation, we developed two methods. Only one of them will be selected as the method of choice.

- 1) Kinematics structure designs
 - a. Grid kinematics structure: Manufacture four connecting spherical joints (unit cells) to form a kinematics structure.
 - b. Hexagon kinematics structure: Manufacture six connecting spherical joints to form a hexagon.
- 2) Stiffness value of the different types of joints in the unit cells based upon experimental results.
- 3) Deformation Method/s
 - a. Analysis of deformation with the grid kinematics structure.¹

¹ After accomplishing these programs for the grid, the analysis can be extended to the hexagon kinematics structure.

1. Method 1: under-constrained, abstract formable kinematics structure model that used the average stiffness value of the joints in the unit cells.
2. Method 2; under-constrained, actual manufacturable kinematics structure model that applies two different stiffness values for the two different types of joint designs that comprise the unit cells.

1.12 Introduction to the Rest of the Thesis

This chapter serves as an introduction to this thesis, explaining the foundation for what will come. Below are brief explanations of the other chapters.

Chapter 2: Literature research about other CAD modeling package

Chapter 3: Design and manufacturing of the kinematics structure

Chapter 4: The math behind the analysis of the deformation of the kinematics structure

Chapter 5: The results from the implementation of the math

Chapter 6: Future works and benefits of this thesis to other members of the Digital Clay

CHAPTER 2

LITERATURE REVIEW

In this chapter are sections discussing about other designs related to the work in this thesis. The first set of sections discuss about other products that work with or apply the “Virtual Clay” idea. The second set that follows will discuss about existing product that is similar to the deformable kinematics structure in this thesis.

2.1 “Virtual Clay” Ideas

Below are examples of existing “Virtual Clay” or products that manipulate “Virtual Clay”.

2.1.1 CyberGrasp and RM-II Hand Master

The human-computer interface idea is a powerful improvement upon the current CAD system and has already provoked others researchers curiosity. Some existing implementations have included a glove-like or haptic manipulator interfaces that focused on reshaping non-physical volumes of ‘virtual clay’ on a computer screen. Examples are the CyberGrasp (Immersion, 2004) and the RM-II Hand Master (Virtual Reality Technology, 2004) as seen in Figure 2.1.



Figure 2.1: CyberGrasp (Left) and RM-II Hand Master (Right)

Applications for these haptic manipulators can be surgical training that requires the physical volumes to behave in a physically based manner (Choi, et. al, 2002). However these sculpting systems were being criticized for relying upon physically based behavior that utilizes multi-scale techniques or pre-computed material properties to achieve real-time performance (Capell, et. al, 2002; Debunne, et. al, 2001; McDonnell and Qin, 2000). It does not look or feel real. Figure 2.2 is an example of a graphically manipulated non-physical volume of ‘virtual clay’ on a computer screen.



Figure 2.2: Face deformation (www.siggraph.org)

These physically based behaviors are often computationally expensive and may lead to unnecessary interaction difficulties. For example, the volume preservation behavior of physical clay is an unwanted and unneeded behavior for our work. Other work in freeform deformation implemented some physically based behaviors (Barr, 1984; Sederberg, 1986) and has utilized a variety of deformation tools (Coquillart, 1990).

2.1.2 The PHANTOM

As haptic interface devices become more popular, the introduction of the PHANTOM by SensAble Technology has spawned a wide variety of applications (SensAble, 2004).



Figure 2.3: SensAble Technology PhanTom

In the area of mechanical product development, physical interaction between user and clay consists primarily of the forces applied by each to the other. In addition, the user can inspect the shape visually and by touching the shape without modifying it. Our approach in our Digital Clay haptic feedback device is to investigate a single mode of interaction to explore capabilities and limitations of tactile interaction, with shape and force feedback through the device.

2.1.3 FreeForm

Another one of SensAble product is the FreeForm modeling system that they are advertising for having real-time force feedback for complex shapes. They are attempting to move into the engineering market of design and manufacturing of products for the user to create various organic shapes as shown in Figure 2.4. Other similar types of haptic

interface devices have also been developed and tested for product development applications (Gurocak, et.al, 2002).



Figure 2.4: FreeForm Modeling for Manufacturing

Although the FreeForm modeling packaged produced very organic shapes on the screen, the shape is still on the screen. The Digital Clay will be a physical device that allows the user to physically view and touch the shape in real life. It will also have a graphic display of the shape on the screen, much like the FreeForm but with an extra plus.

2.1.4 Feelex

It appears that so far every product can manipulate shape on the screen, but not in physical life. With the introduction of the Feelex by the Virtual Reality Lab (VR Lab) in Japan, the user can now feel the haptic feedback of any given shape with their bare hands and manipulate the shape without any additional hardware as seen in Figure 2.5 (Iwata, et.al, 2001).



Figure 2.5: Feelex Version 1 and 2

This innovative product is the next step into providing haptic force feedback for user without using any extraneous devices. The digital clay device will take that idea to become a more advanced version of the Feelex in force feedback device. It will not be just a bunch of pins that moves up and down like an animated pin-cushion but a series of bubbles that inflates and deflates upon applied force. This will allow greater shape deformation and resolution. Plus the digital clay device is a combination of both the SensAble technology's Freeform and the VR Lab's Feelex, advancing what already exist. Digital Clay-- so far nothing is like it in the current market. That's what makes this project and thesis a challenging and rewarding experience.

2.2 Elastic Deformation Products

Previously we gave examples of different products that would allow the user to deform CAD models using force-feedback mechanism. Now instead of discussing about

the whole Digital Clay device by giving examples of competitive products, lets us describe the focus of this thesis: the deformable kinematics structure. This section will describe various existing products that use elastic deformation as the source of motion. In all the examples below, each product is a 2D deformable device. Their designs will help develop the 3D deformable crust matrix.

2.2.1 Compliant Mechanism

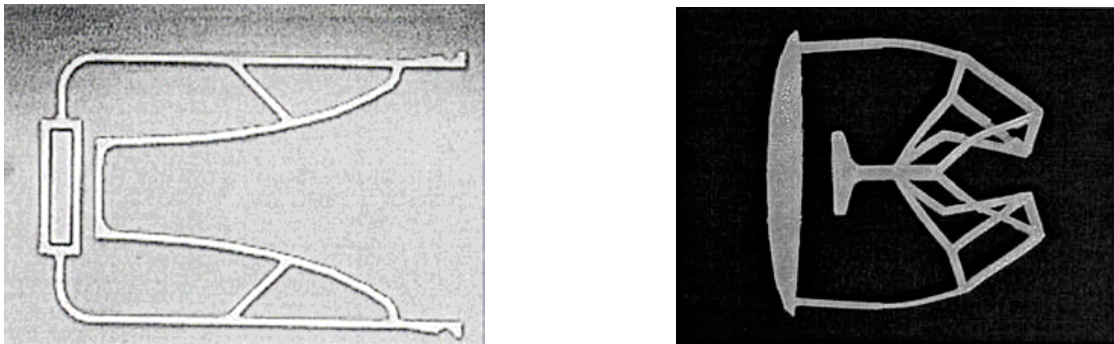


Figure 2.6: Compliant Crimping (Left). Compliant Gripper (Right)

Designed by a team of Mechanical Engineers from the University of Michigan, Ann Arbor and Sandia National Laboratories, compliant mechanisms are single-structure mechanisms that can transmit motion through flexible hinges (Kota, et.al, 2001). These mechanisms consist of connecting rigid links with elastic deformable joints as seen in Figure 2.6 with both the Compliant Crimping on the left side and the Gripper on the right side. As the two handles on the far right of the Compliant Crimping are pushed together, the rectangle on the far left will move horizontally to the left. For the Compliant Gripper,

as the handle in the center is pulled to the left, the two trapezoids on the far right will be pulled together as if it was pinching something.

Similarly this thesis is attempting to create a design where a single action will create a series of reactions to accomplish a task. Currently we are investigating compliant mechanisms for ideas to develop our Digital Clay deformable crust.

2.2.2 Flexural-Based Gripper

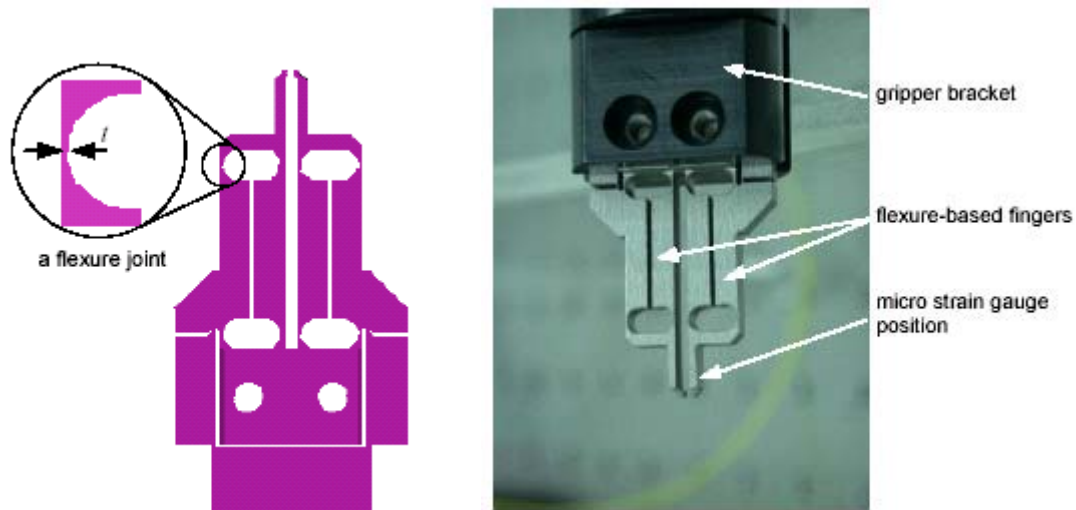


Figure 2.7: Flexural-Based Gripper Design (Left). Manufactured (Right)

Comparable to Kota's Compliant Mechanisms, Chen and Lin's Flexural-based Gripper applies the elastic deformation capability of the material to create motion with little or no assembly necessary (Chen, 2002). In this case, this Flexural-based Gripper is used for handling optical fibers. In the left image of Figure 2.7, the zoom image shows the curved surface and the thin walls of the design that would act as the flexible joint for

the mechanism and the right image shows the gripper being manufactured. This will be another idea that would help how our Digital Clay deformable crust becomes more of a reality.

Both of these 2D compliant mechanism designs set the stage for developing the Digital Clay crust matrix with compliant hinges that would use elastic deformation as a source of motion to deform in 3D.

2.3 Ending Comment

Based upon this literature review and the introduction chapter, the design expectation for the Digital Clay crust matrix is to design a deformable crust as a physical mesh that would respond to human touch. In the introduction chapter, we mentioned that the Digital Clay device should also receive signal from the computer to deform. This set another requirement that there should be some sort of interconnection between the actuators and the display device. However at this stage, we need to design the crust matrix that would deform with consideration for actuators and sensors than to design the interconnection. The next chapter describes the various designs that the deformable crust matrix went through before we settle down on analyzing one design.

CHAPTER 3

THE SKETCHBOOK OF MATRIX DESIGNS

The design of the deformable infrastructure that would generate manifolds of shapes went through various iterations and modifications. The final design is a deformable crust of spherical joints. Before the spherical joint was developed, the parts that make up the crust matrix are generically named “unit cells”--individual cells that can deform and can be combined to deform as a whole. Below will be explanations of the design process similar to Beitz and Pahl Design process (Beitz and Pahl, 1996) and the manufacturing of different concepts for the unit cells.

3.1 Requirement List

After knowing what is expected from this matrix based upon the previous chapters, the requirement list will be created to ensure that the customer’s demands are being met as well as any requirements that we, the designers, may have. The customers are the Digital Clay team members. A requirement list is a design specification list that states which features or characteristics of the subject of study are either demands (“[features] that must be met under all circumstances”) or wishes (“[features] that should be taken into consideration whenever possible”) (Beitz and Pahl, 1996). The people involved in fulfilling the requirements are various people who are/were classmates, project members, and Digital Clay team members. I am the principle designer who will gather the necessary data and guide the design development phase during this whole process. Below are the original descriptions and specifications for one unit cell design

that we are developing to satisfy our customer. While creating the list, we will also consider the matrix of unit cells. As previously stated, the unit cell should be able to form a deformable kinematics structure by linking together and becoming a matrix of unit cells. During this design stage, the matrix of unit cells is not attached to any mechanical or electrical devices. The energy for deformation of the crust matrix will be human powered. Later in the design stage, the energy of deformation will be powered by mechanical or electrical devices. It is beyond the scope of this thesis to go into the details of the devices. Below is Table 3.1 that describes the requirements for design these unit cells that would link together.

Table 3.1: Requirement List

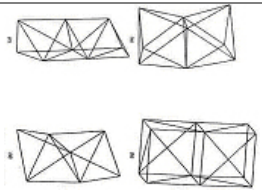
Problem Statement:		Schematic:
<i>Design a unit cell that is capable of deforming in various directions without breaking and able to link together to form a matrix of the cells.</i>		
D	Requirements	
W		
	1. Geometry	
W	Width of unit cell: width <18mm (a fingertip width)	
W	Depth of unit cell: Depth <18mm (a fingertip width)	
W	Height of unit cell: Height <18mm (a fingertip width)	
D	Unit cells capable of attaching to neighboring cells to form a matrix structure.	
D	Matrix of cells is scalable	
	2. Kinematics	
D	Cell Deformable angle: 90 degree - 180 degree	
D	Matrix constructed from these cells has to be able to deform	
D	Matrix constructed from these cells must be portable	
	3. Forces	
D	Cell deforms in three directions (x, y, z)	
D	Applied forces on matrix can act in any direction	
W	Applied force for matrix deformation: 3 N - 6 N	

Table 3.2: Requirement List (continued)

D	Requirements
W	
	4. Energy
D	Mechanism of deformation: human power for shape input
	5. Material
D	Material capable of deforming
W	Material is elastic
D	Cell withstands repeatable deformation: 20 count - 500 count
	6. Safety
D	Operator safe
	6. Ergonomics
W	Matrix of cells is smooth to human touch
	8. Production
W	Manufacturing cell: SLA.
	9. Assembly
W	All cells molded as one piece.
D	Matrix is constructed of multiple cells.
	10. Operation
D	Human touch on matrix causes deformation.
D	Computer actuated when matrix is connected to computer
	11. Maintenance
W	Easy exchange of cells within matrix
	12. Recycle & Environmental
W	Environmentally safe material.

3.2 Check (Clarifying The Task)

With the completion of a detailed requirement list that focuses on our customer's needs and wants as well as ours, Phase II, conceptual design, is next. The reason for this is that we now have a list that lays out the functions and requirements that are necessary for our design to be successful.

3.3 Abstracting to Identify the Essential Problem

After developing a requirement list according to our customer's demands and our wishes, we begin to abstract the conditions attributed to the problem and task, trying to venture away from any design fixations. To accomplish this task, abstraction and problem formation are done using the five-step method.

3.3.1 Abstraction and Problem Formation and Systematic Broadening

From the Beitz and Pahl Design process the Abstraction and Problem formation process is a five-step process that goes through the requirement list and reduces the list to one main problem formation statement. The process consists of:

- I. Eliminate personal preference
- II. Omits requirements that have no direct bearing on the function and the essential constraints,
- III. Transform quantitative into qualitative data and reduce them to essential statements
- IV. Generalize the statement made in step III
- V. Formulate the problem in solution-neutral terms.

This process was already performed in ME 6101: Design Engineering. The result is **Design a multi-connected, scalable cell that can deform in various directions.** Upon developing this solution-neutral problem statement, we further broaden the project to

prevent any potential design fixations. Systematic broadening is done by abstracting from a specific statement to a general statement.

For Systematic Broadening, this process was already preformed in ME 6101. The final statement is: **Design a scalable matrix that deforms.**

3.4 Function Structure

As a result from systematic broadening, our portion of the project is focusing on just the construction and deformation of a cell that would later become the matrix. The function structure is seen to be relatively simple.

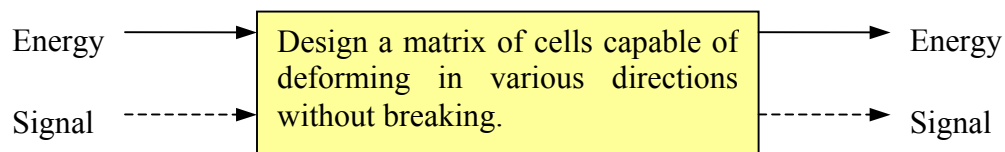


Figure 3.1: Overall Function Structure

The energy input is the amount of force applied to the unit cell matrix to deform it. Meanwhile, the signal represents the direction and location of the applied force. For instance, the various locations of the applied force will result in different visual deformation signals. Figure 3.2 is a more descriptive function structure. When the energy and the signal are given to the matrix, the matrix would respond and change the positions of various unit cells. Because the energy gets lost when the applied force is moved, not every unit cells receive the same amount of energy; therefore not every unit cells will deform the same amount. When the residual force is removed, the unit cells will return to their original shape.

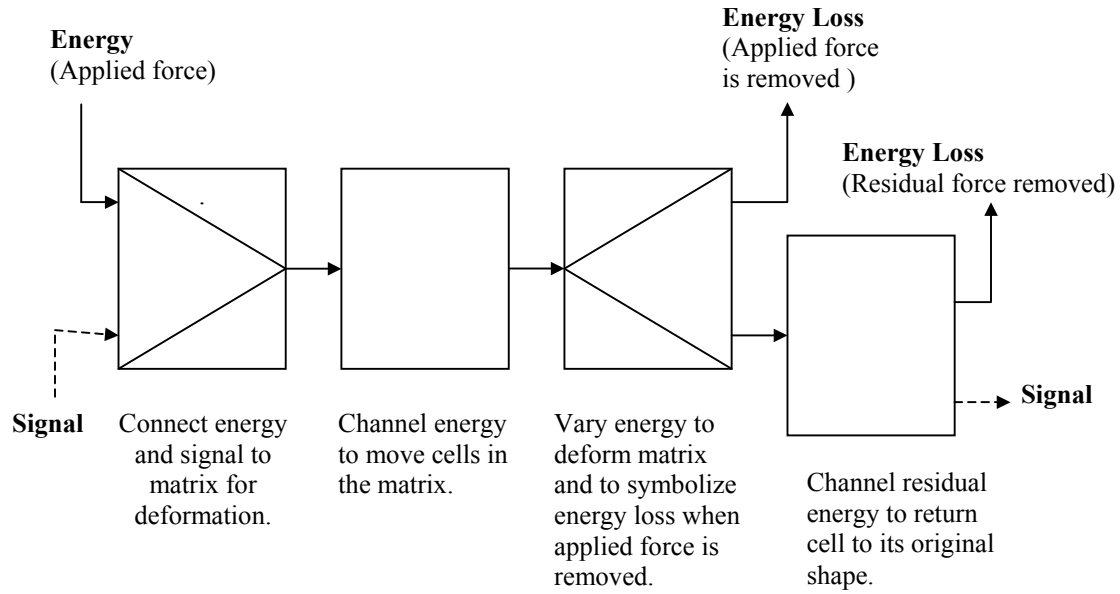


Figure 3.2: Function Structure Using Generally Valid Functions

3.5 The Manufacturing Technique

There are various ways to create prototypes of the Digital Clay matrix for testing and evaluation. Below are several techniques that are being considered.

3.5.1 MEMS

Micro-electromechanical systems (MEMS) technology is an integration of sensors, actuators, and electronics on a common substrate using micro-fabrication technology (MEMS and Nanotechnology Clearinghouse Website). Below are various MEMS techniques being considered for prototyping the Digital Clay matrix.

3.5.1.1 Thermal Press Molding

Using a mold made of aluminum or stainless steel, the Digital Clay matrix can be formed by pouring a thermo-set polymer onto the mold. The polymer would then solidify by applying heat and pressure. The materials being investigated are Dyneon Elastomer and Polyethylene. This process seems promising. We may create a mold for the matrix in the future. It is beyond the scope of this thesis to discuss the possibility of this mold.

3.5.1.2 Injection Molding

Similar to the thermal press molding, a mold is first created. The selected polymer is heated to a quasi-liquid state, then injected into the mold using a vacuum and cured by heat. This process is not successful because the walls of the Digital Clay structures are too thin and complicated for the injection molding process to work properly.

3.5.1.3 PDMS Cast Molding

Poly-dimethylsiloxane (PDMS) is a liquid pre-polymer that is cast against a mold. After curing, the cross-linked and elastomeric PDMS is carefully peeled off from the mold. The surface of the cured PDMS is the structure of the Digital Clay matrix. PDMS has an excellent capability of capturing details, but the material is too flexible for our needs.

3.5.1.4 Lamination

Another MEMS approach is using the lamination-based polymeric approach that bonds to substrates by heat and pressure. In these approaches, analogous to lamination-

based electronic packaging approaches, individual sheets of material are lithographically patterned or laser-cut to form the required chambers and fluidic interconnects, and then are laminated together to form the final structure (Dalmia, 2002). However cost, shape generation capability, dimension issues, and facility availability are issues when using this technology.

3.5.2 LCVD

One rapid prototyping technique being considered is Laser Chemical Vapor Deposition (LCVD). A laser CVD rapid prototyping system is one of the promising manufacturing techniques that is under development in the School of Mechanical Engineer at Georgia Institute of Technology. The process has the capability of fabricating complex net-shaped metallic and ceramic structures by depositing powder using laser to heat a heated substrate (Park, 2003). LCVD can satisfy several of the demands from the matrix requirement list because the process deposits material at the atomic level, producing a material that is fully dense, ultra-pure, and mechanically sound. Since LCVD can also produce fibers or layers in any given direction, the design of the crust matrix and the building orientation will be not restricted by this technique. Furthermore, a capacity for multiple materials permits composite structures and functionally-graded materials and alleviates traditional material restrictions imposed by a given prototyping technique (Lackey, 2002). LCVD is a promising manufacturing technology that may be beneficial to Digital Clay, however it is a new process that is still being investigated and may not be available to the manufacturing community until later in the future.

3.5.3 Other Techniques

There are other standard techniques that can be used to manufacture the matrix. One included the injection molding using a low viscosity liquid with a low cooling rate to fill up all the small spaces and holes of the crust matrix. Another technique is using an open face molding with a spray adhesive and a stamp cutter. Both of these techniques are promising, but expensive to create the mold. At this time we are searching for a technique that has a fast turn-around time with a high accuracy result.

3.5.4 SLA

Rapid prototyping technology using Stereolithography (SLA) technique has a high turn around rate with high accuracy. Because of this feature, using SLA technology will allow us to vary the dimensions of the Digital Clay cell and build the matrix within hours to meet any specific task that our client may want to use the product for. Furthermore, it will satisfy the demand imposed by our client that the Digital Clay cell is scalable. The cells will have to be integrated together to form a matrix that will respond to at least a finger width (approx. 18 mm) of applied pressure. With rapid prototyping technology, multitude of thin and small cells can be generated at a low cost with relatively fast results without supervision. Therefore a rapid prototyping method using a stereolithography machine appears to be the most efficient method of creating our Digital Clay cells and for that reason it was placed as a wish on our requirement list.

Refer to Figure 3.3 below for a brief definition of Stereolithography. In Stereolithography, solid objects are created by using a layer based manufacturing

technique. First the designer would create a CAD model of the objects, in this case the Rook from the Chess game. Next, a computer will “slice” the CAD model into cross-section contours, one on top of the other. Third, the stereolithography machine will create the support structures for levitating the Rook above the platform that the Rook will be resting upon. Fourth, the slices are created by tracing the 2D contours from the CAD models in a vat of photopolymer resin with a laser. Each slice is created when the platform that the parts rested on is lowered into the vat, exposing only a thin layer of resin to the laser at any one time. The final step is cleaning, post-curing, and detail finishing the parts (personal preference of the designer). At the end we will have our Rook! For more information about stereolithography, refer to “Rapid Prototyping & Manufacturing: Fundamentals of Stereolithography” by Paul Jacob (Jacobs, 1992).

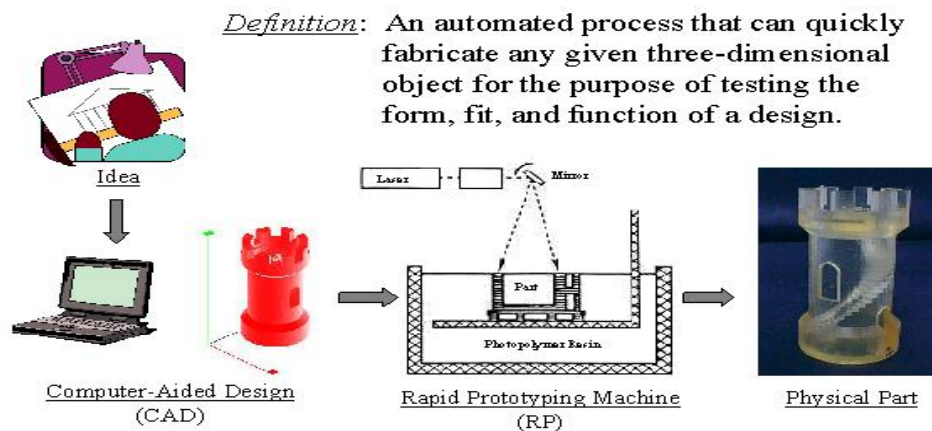


Figure 3.3: An Additive Fabrication Process – Stereolithography (Jacobs, 1992)

3.6 Design and Manufacturing of Designs

From the requirement list and the function structure, several ideas evolved. Below are some of the ideas. Some of these were manufacturable while others will crash due to the part designs exceeding the machine capability.

3.6.1 The Flexible Corners

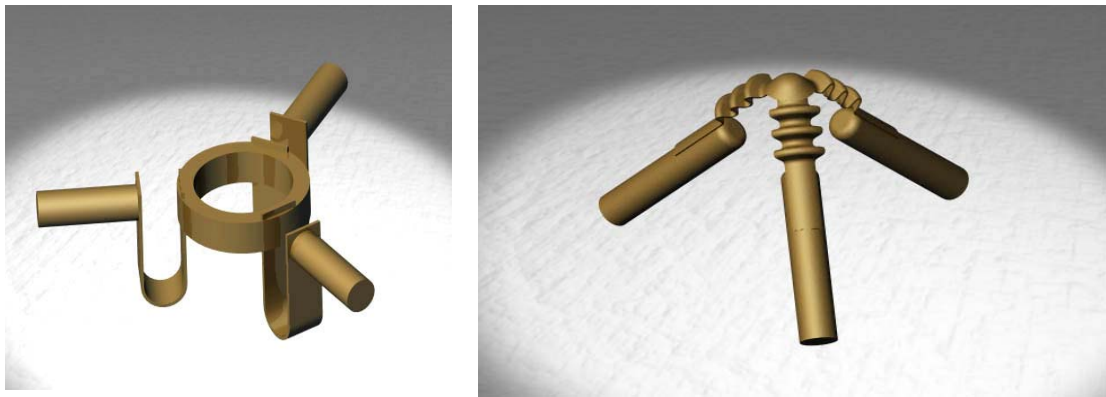


Figure 3.4: The Flexible Corners

In Figure 3.4, these corners can connect to other corners of the same design and create a matrix of flexible corners. The shapes are simple enough to be scaled down without losing much detail. However these shapes poses problem when they are rapid prototyped. The flexible joints would not build properly because of the thinness of the joints. After resolving the manufacturing problem, the joints would break after bending them less than 10 times by hand. Because of this problem, the flexible corners idea was eliminated.

3.6.2 Deformable Cubes

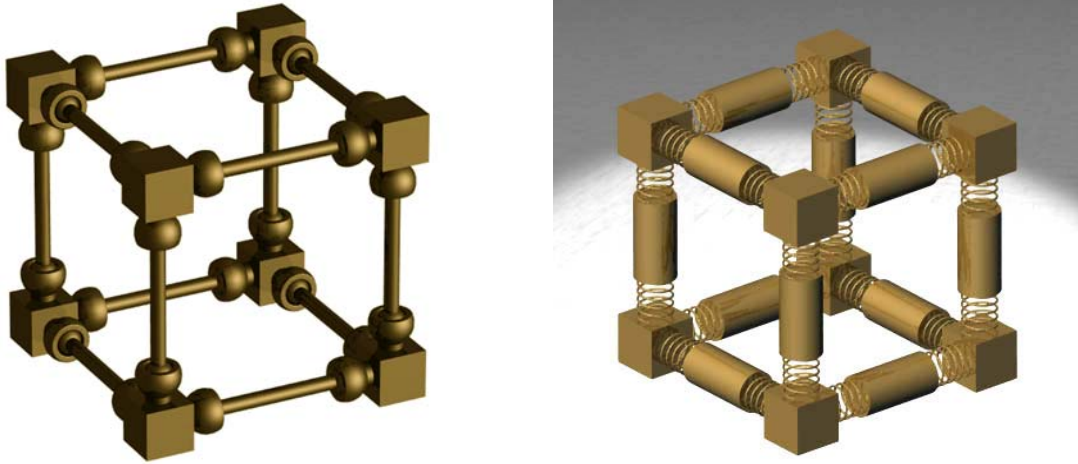


Figure 3.5: Deformable Cubes

The deformable cubes are actually a variation of the flexible corners in Figure 3.5. The cube on the left has several ball and socket joints connected to a square base. The balls rotate perfectly within the socket and are able to deform as a whole. However the angle of deformation is based upon the opening of the socket. As a whole, the cube deforms around 20-30 degrees.

The other cube on the right has a greater degree of freedom because of the springs. During manufacturing, this cube crashes more than any of the other designs. Another minus point is that as this cube on the right is scaled down, the deformation capability decreases.

Of both designs, scalability is a big question. Will the matrix of these cell deform as well as it can when the cells are scaled down?

3.6.3 Compliant Hinges

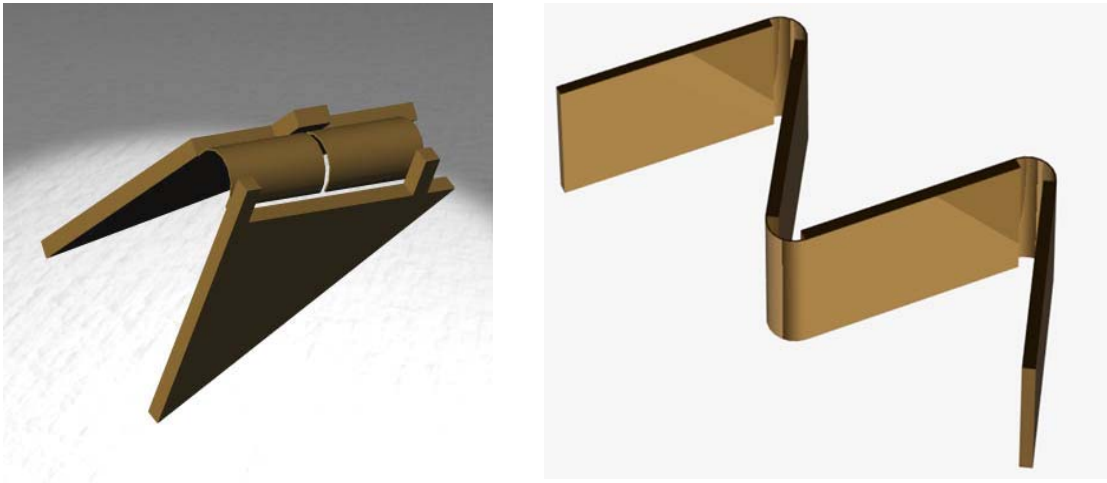


Figure 3.6: Compliant Hinges

The compliant hinges are two plates connected by a thin flexible plate. The one on the left is a modification of Jacob Diez's compliant hinges from his robotic hand(Diez, 2001). The design on the right is a modification of the one on the left to improve the fatigue life. Both of these designs will deform upon applied force and return back to original shape after the force is released. The one on the right is one of the easiest designs to scale and manufacture due to its simplicity. The downfall is that it only deforms in two direction: Z and X or Z and Y.

3.6.4 Deformable Crust Design

While developing a feasible manufacturable joint design, crust design evolved with the help from Paul Bosscher. The crust design is a deformable matrix that acts like a piece of cloth. As the fluid flows in from the valves, the crust will deform and take shape from the applied pressure, as it was a piece of cloth.

Figure 3.7 demonstrates the crust deformation idea. Each vertex is an abstract representation of a deformable unit cell and each line is a connecting rod from the unit cell.

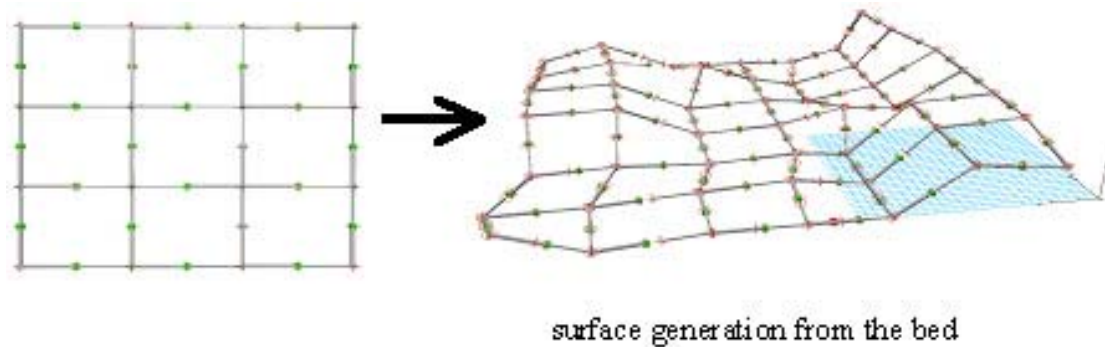


Figure 3.7: Deformable Crust Concepts

3.6.4.1 Unit Cell for Crust

The crust idea seems simple and effective enough for fulfilling the requirement list. The challenge with crust designs is their Manufacturability. The construction of spherical, revolute, or other kinematics joints at small size scales is difficult. To duplicate the behavior of spherical joints, we can use a collection of links and revolute joints, where the joint axes have a common intersection point (Bosscher, 2003) as one

will see in the following sections. Below are ideas for developing the unit cells that would compose the crust matrix.

3.6.4.1.1 Eight-Sided Unit Cell

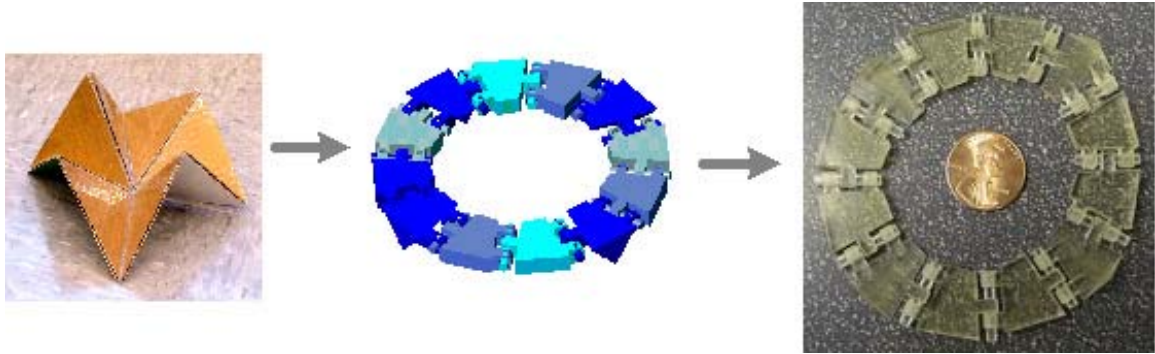


Figure 3.8: Eight-sided Deformable Crust

From paper to model to manufactured design concept, Figure 3.8 describes a unit cell with eight sides and eight revolving joints. The greatest complication in CAD modeling is assembling the individual pieces together and scaling the cell down. After prototyping in SLA, the cell is very deformable but not very rigid. It deforms like a piece of cloth, but the joint clearances cause repeatability problems.

3.6.4.1.2 Spherical Joint Unit Cell



Figure 3.9: Spherical Joint Unit Cell

This unit cell consists of eight expandable faces with a revolving joint in between each face. There are also 4 linking faces with two revolving joints that are connected to the expanding faces. This gives a total of 12 faces and 12 joints. In this case, all the faces are triangles. The expanding faces are nicknamed “intermediate triangles” because they are in-between the smaller faces. The smaller faces are nicknamed “linking triangles”, because they will be used for linking to the next neighboring spherical joint. The CAD modeling of this unit cell has the same difficulties as the Eight-sided Deformable cells with assembling and scaling of the cell. In the manufacturing aspect, the cell deforms as well as the Eight-sided Deformable Cell. The difference is that there are more degrees of freedom because each linking triangle rotation capability is not affected by the other linking triangles. The triangles in the Eight-sided Deformable Crust have more inter-connection and less Degrees-of-freedom. The Spherical Joint unit cell has more potential for deformation but still does not provided the resistance force for feedback that was requested in the requirement list.

3.6.4.1.3 Linear Triangles

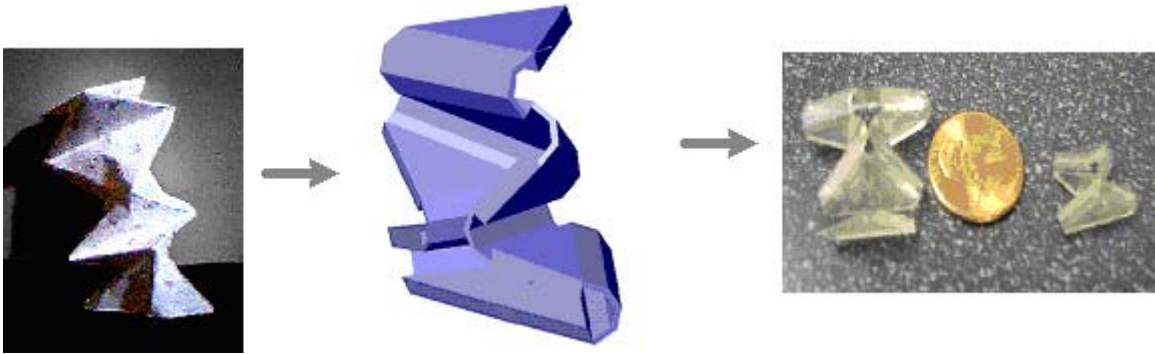


Figure 3.10: Linear Triangles

Unlike the previous two unit cells for the crust design, this design utilizes the compliant joints mechanism from prior designs in section 3.7.3 Compliant Hinges. CAD modeling is less complicated with no assembly necessary (it is drawn as one piece). The CAD model is also less complicated to scale down. During manufacturing in SLA, this design is the most stable of all three designs. However, the manufactured piece deforms the least of all three crust designs due to interference and neighboring walls.

3.6.5 Selection Process of the Unit Cells for the Crust Matrix

The most promising design needs to be selected from the manufactured prototypes of the unit cells for the crust matrix. The best design would be based on three criteria--manufacturability, scalability, and dynamic functionality. Each criterion is rated a scale of 1 (worst) to 10 (best). A detailed description corresponding to each rating is shown in the tables below.

Table 3.3: Manufacturing Attribute

Description	Rating
<i>Very Simple</i> - Easily manufacture without any complications at a quick pace.	10
<i>Simple</i> - Easy to manufacture with minor complications.	7
<i>Complex</i> - Manufacturing takes time and potential problems are encountered.	4
<i>Very Complex</i> - Manufacturing is extremely difficult and time consuming. Problems constantly have to be overcome.	1

Table 3.4: Scalable Attribute

Description	Rating
<i>Very Scalable</i> – Cell is scalable without any problems and performs function to perfection	10
<i>Slightly Scalable</i> – Cell is scalable but with minor problems and reduced performance.	7
<i>Normally Scalable</i> – Cell is not very scalable and has some problems with low performance.	4
<i>Un-scalable</i> – Cell is hardly scalable and has serious problems with major performance issues.	1




Table 3.5: Deformation Attribute

Description	Rating
<i>Very Dynamic</i> - Cells perform desired deformation to utmost perfection.	10
<i>Slightly Dynamic</i> - Cells perform desired deformation with slight interference or other problems.	7
<i>Nominally Dynamic</i> - Cells have marked problems and do not perform full deformation.	4
<i>Non-Dynamic</i> - Cells are virtually un-deformable and do not achieve the required motion at all.	1

Table 3.6: Rank of the Criteria

Description	Rating
<i>Dynamic Function</i> – Ability of the cell to simulate desired motion	3
<i>Manufacturing</i> – Manufacturing of the cell without causing any problems	2
<i>Scalable</i> – Ability of the cell to be put in a matrix.	1

Table 3.7: Design Selection Table

Criteria			
scale	$5 * 1 = 5$	$5 * 1 = 5$	$9 * 1 = 9$
Dyn	$6 * 3 = 18$	$9 * 3 = 27$	$1 * 3 = 3$
Manuf	$6 * 2 = 12$	$4 * 2 = 8$	$10 * 2 = 20$
Total	35	40	32

After considering the scalability of all the models, the parts were manufactured (repeatedly in some cases) and then tested dynamically. The highest importance is attached to the deformation functionality. After several tests, it was found that Spherical Joint Unit Cell model best simulated the desired motion. In addition, it was decently scalable and can be extended to form a matrix of cells. The only problem encountered is in manufacturability. Due to the vertical alignment of the model, the support trusses that are automatically formed during the manufacturing process in SLA are not able to support the structure and hence the model is hard to manufacture. The model still is considerably superior over the other models because of its better dynamic functionality.

3.6.6 Matrix Selection

Although the unit cell was selected based upon the given criteria, it is not yet certain that the crust matrix would deform, as one would want it to base upon the quality of one unit cell. Therefore it is necessary to consider the unit cell as part of a whole matrix before coming to a decision that the design selected is best fitted for the task given. At this point of time, it is unnecessary to attempt building different matrices with different unit cells. The selection process given is only meant to be a suggestion of the unit cell design that should be carried on to the next step. While going through the design process stage, we can modify the unit cell so that it can create a matrix. During the process we would also consider the other designs that did not arise high.

Below are suggested criteria for the selection process among various matrices with different unit cell designs.

Connection Capability:

Since each unit cell is connected to each other, how are they connected? Is the connection robust enough to handle various deformations and the added stress to the displacement?

Manufacturability:

Although the selected unit cell is manufacturable, the matrix may not be as manufacturable. There maybe interface problem as the unit cells are connected together.

Scalability:

Can the matrix be scaled down? Although the unit cells individually scalable, it does not mean that the whole matrix is scalable. Perhaps the connection between unit cells would prevent the matrix from properly being scaled down while still maintaining deformation capability.

Deformability:

As the displacements are added to various points on the matrix, how much can the neighboring cells deform? How much displacement can any one unit cell handle before interference or structure damage occurs?

As mentioned above, these new criteria given are for comparing various matrices built using different unit cell designs. Currently we are only working with one type of

unit cell with different matrix designs. In the future after developing more matrices using different unit cell design, these new criteria can come into play.

3.6.7 Modification of Selected Unit Cell

From the previous section, spherical joint unit cell is selected to progress further into the design development stage. However the manufacturing issue is a problem. From the previous section, the least problematic is the linear triangles with the compliant joints. In this section, the two strong qualities from the two previous designs will be combined into one: spherical joint with compliant joints. From the idea of the spherical joint unit cell connecting together to form a square grid, the spherical joint unit cell can also be modified for three linking triangles to form a hexagonal grid matrix. There are other design modifications such as applying prismatic joints between two connecting unit cells (Bosscher, 2003). It is beyond the scope of this thesis to go into the details of these modifications. Below are the modifications that are being considered and the development of the matrix.

3.6.8 Matrix

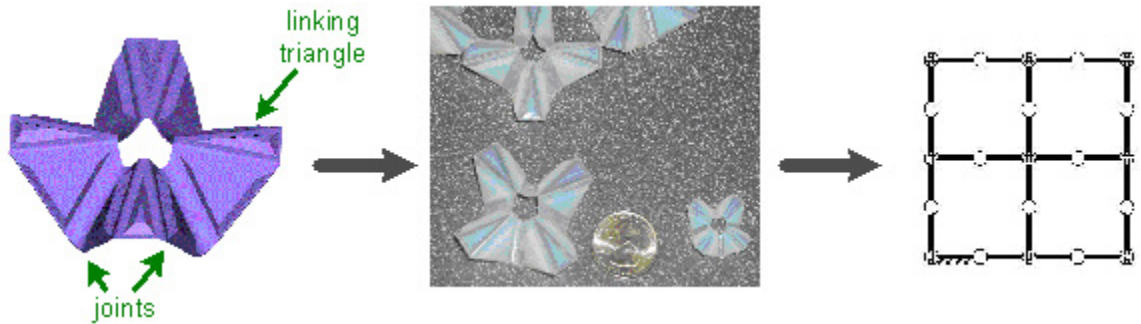


Figure 3.11: Grid Matrix Unit Cell

Figure 3.11 shows the spherical joint unit cells from the previous design being modified with compliant joints. Only two of those joints are labeled in the first image. Another modification is that the linking triangles are pyramids to add rigidity and creating the coupling effect when the unit cells are linked together. Because of this design modification, the spherical joint unit cell can be scaled down as seen in the middle image with the quarter. From this design the cells can be linked together as seen in the last image as vertices with dark circles and matrices as seen in Figure 3.11.



Figure 3.12: Grid Matrices

Figure 3.12 shows the unit cells being linked together to create a grid. The first image shows circular rods in between each unit cells. The rod was originally thought to

expand the length between the unit cells while maintaining the rigid linking. However, the rods twist and bend due to the material used. The material is DSM Somos 8120 photopolymer resin. Other resins were explored, but the DSM Somos 8120 has the most promising material characteristics. This material provides the needed flexibility for the compliant joints to function properly and also the stiffness for the linking triangles to create the coupling effect when the unit cells are linked together. The second image shows that the rod length is reduced when the unit cells shrunk. The last image shows that the rods are completely eliminated from the matrix with two unit cells almost equaling the diameter of a penny. A four-by-four matrix with these unit cells is less than the size of a business card. The deformation capability of the grid matrix is analyzed in the next chapter.

3.6.9 Hexagon Matrix



Figure 3.13: Hexagon Unit Cell

Similar to the grid design, the hex matrix has three linking triangles instead of four and each unit cells are connected together to create a matrix of hexagons. Again the revolving joints that were originally part of the spherical joint unit cell design are being

replaced by compliant joints as seen in the first image of Figure 3.13. As previously stated this modification allows the unit cells to be scaled down as seen in the middle image with the penny and connected as seen as vertices with the darker circles in the last image and in Figure 3.14.

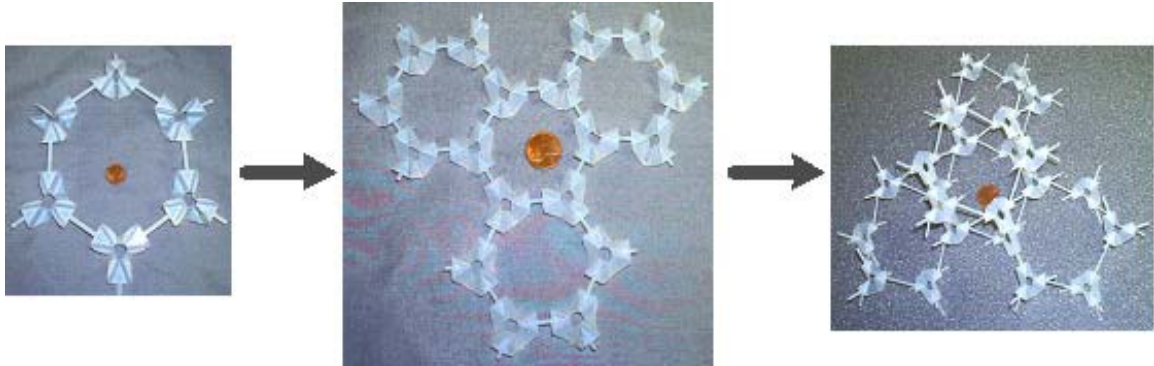


Figure 3.14: Hexagon Matrices

In Figure 3.14, the rod idea was tested again. Again, the rod idea is slowly being eliminated from the design. The hexagon unit cells did not follow the same path in design modification as the grid matrix as seen in the previous section. The reason is the center voids of the hexagon matrix. In the middle image, a penny occupies one of the center voids of the hexagon matrix. As the matrix deforms, the center voids will prevent the matrix from creating a continual surface that we would expect the matrix to create when the matrix deforms. The last image is two matrices overlapping to attempt eliminating the jaggedness of the shape generation from the hexagon matrix. Offsetting and overlapping modification was also considered. This offsetting and overlapping did reduce the center voids, but it also reduces the deformation capability of the matrix.

3.7 Crust Matrix MEMS Style

The previous sections described the Digital Clay crust matrix built using SLA technology. Earlier, we mentioned the possibility of using MEMS technology to fabricate the crust matrix. The introduction chapter also mentioned the possibility of adding bubble actuators in the joints of the crust matrix, expanding and compressing the joints by filling and draining the fluids in the bubbles. There may be additional or different types of considerations when MEMS technology is being applied instead of SLA. Since I am not an expert in MEMS, I will pose several questions to the people who will attempt to create the crust in MEMS with the bubble actuators.

Questions:

- 1) What is the material of fabrication?
- 2) From the material of usage, what is the stiffness of the joints?
- 3) Will there be enough force feedback to maintain shape deformation?
- 4) Can the unit cells be rigidly connected through their linking triangles while the joint maintains its flexible capability?
- 5) Will the curvature of the joints change as the fabrication process changes from SLA to MEMS?
- 6) How small can we get and still be able to add actuators to each unit cell?

These questions may evoke some creative juice in the MEMS department. Currently, they have developed several ideas to fabricate and interface a bubble-like actuator in the joints. The pictures of several different concepts for the bubbles are shown below.

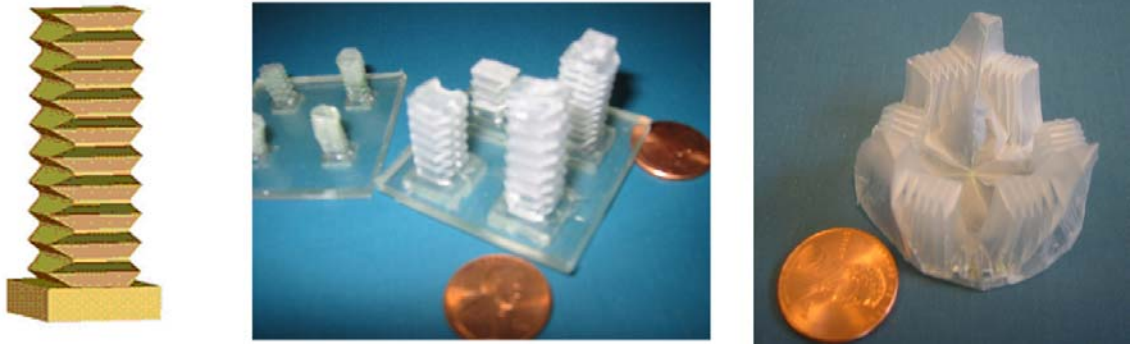


Figure 3:15: Bellows Bubble Actuators Concept 1



Figure 3:16: Bellows Bubble Actuators Concept 2

It is beyond the scope of this thesis to explain these bellows actuator designs.

Another idea for designing the bubble is to enclose the joints using a membrane that can deflate. At maximum inflation, the joints are at their largest angle of deformation. Figure 3.17 shows one unit cell that the bubble has to work with. Figure 3.18 shows two images describing the membrane enclosing one joint. The first image describes one joint being deflated. The next image describes the joint at the maximum deformation.

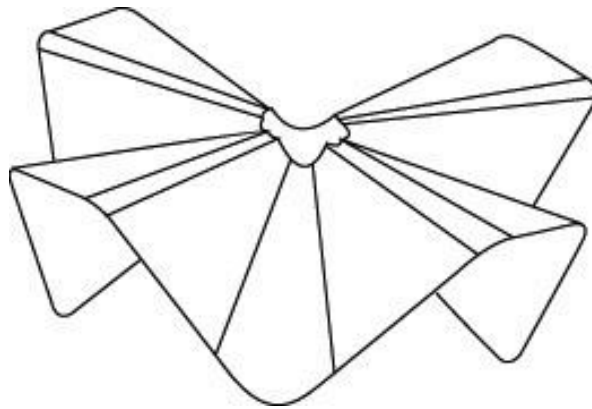


Figure 3.17: One Unit Cell

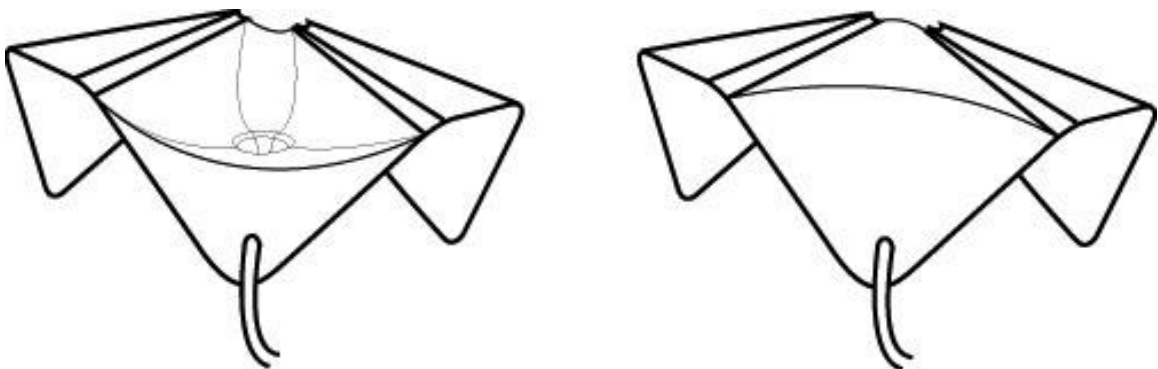


Figure 3.18: Enclosing Membrane

The tube coming off the front is where the fluid will be flowing from. Currently this idea is being criticized because the membrane needs to be expandable like a rubber band. If the membrane is being built at the same time as the crust matrix with every joint at its maximum deformation, the matrix will contour into various different shapes. This may cause problem during the manufacturing process. Currently Sharon Wu in the MEMS department is investigating possible elastic materials for manufacturing the bubbles. Another possibility is that the membrane is built deflated. When the crust is at its resting state, there will not be any force applied to the joints. When it expands, the fluid pressure being applied will have to be greater than the force created by the stiffness in the joint and the membrane. The same situation occurs when the joint compresses.

Fabricating the crust matrix using MEMS technology while designing the bubble actuators for the joints is a difficult task that requires extensive research in design exploration, manufacturing techniques, and material study. The MEMS department is studying all three areas. Once this is accomplished, we will have the interconnection between the crust matrix and the rest of the hardware. Since this is not the focus of this thesis. We will not continue discussing this interconnection.

3.8 Ending Remarks

From the two crust designs, grid matrix design is selected to progress further into the design stage. The reason for this selection is because the grid matrix has the best combination of functionality, scalability, and manufacturability. Also the center voids of the grids are smaller than the hexagon, producing smoother surfaces.

The hexagon matrix is an excellent idea because each joint of each unit cell has a greater angle of deformation than the grid design. The calculation of Degrees-of-Freedom for both the hexagon and the grid matrix shows that the hexagon matrix has a greater capability to create shapes (Bosscher, 2003). However, we need to find a manufacturing technique that can produce the hexagon matrix small enough to reduce the center voids, which in turn reduces the jaggedness. The possible solution is using the DSM Somos 8120 resin in the 3D system SLA VIPER machine that is designed specifically for creating micro products. However as the CAD model of the unit cells (both grid and hexagon) is scaled down, the mathematics behind the CAD model will increase in complexity to maintain the relationship among the features in the model. This may cause problems when the facets of the models do not align properly. In turn, it will create holes and unwanted articles in the models themselves. In a chain reaction, the results of the manufactured CAD models will be either unacceptable or failed parts. It is beyond the scope of this thesis to further investigate these phenomena.

Another reason for choosing to progress further with the grid design is the geometric shape of the grid design. It is easier to analyze a square than a non-right angle hexagon. Once the analysis of the deformation for the grid is accomplished, the mathematics can be extrapolated to the hexagon. The next chapter describes the math.

CHAPTER 4

IT IS THE PRINCIPLES BEHIND THE MATH

This chapter will analyze the kinematics of the grid matrix developed in the previous chapter. The final grid matrix design, comprised of modular unit cells, is shown Figure 4.1.

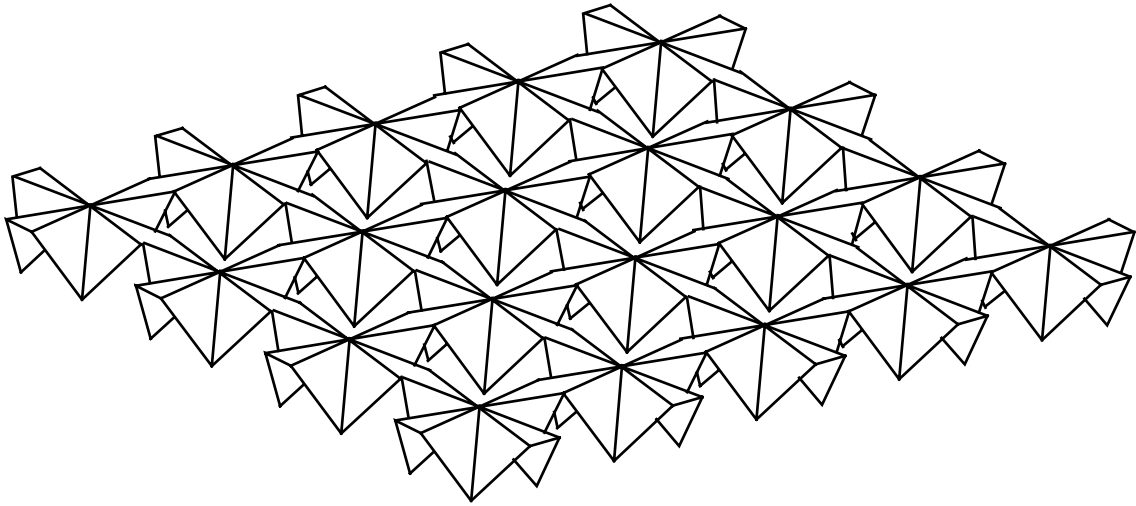


Figure 4.1: Crust Matrix

The matrix consists of rigid connections between each unit cell. If one unit cell moves, the connecting unit cells will deform as well. Therefore there is a coupling effect inherent in the structure's deformation, as shown in Figure 4.2.

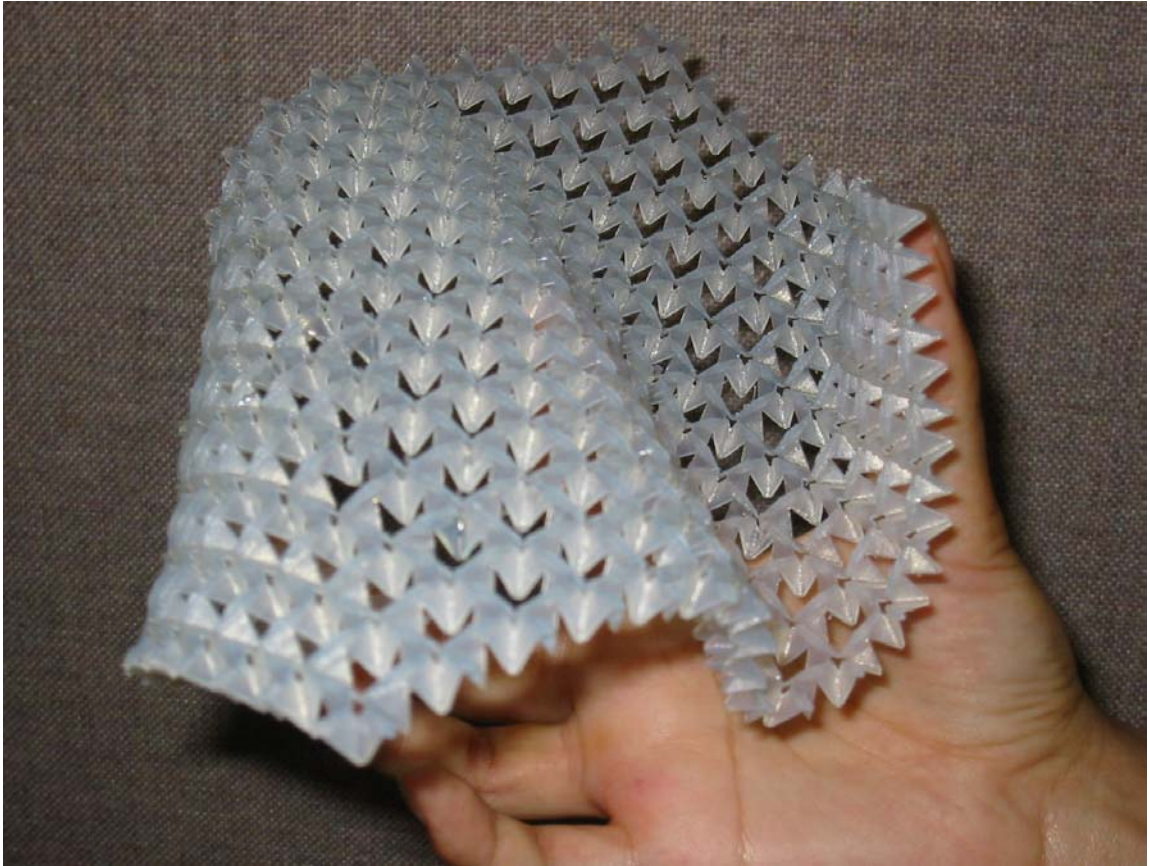


Figure 4.2: Actual Crust Matrix Deforming

The shape in Figure 4.2 is at the lowest energy state of the matrix based upon the external inputs and geometric design of the matrix. In this case the external inputs are the pressure of the fingers applied at various locations. The geometric design consists of the individual unit cells that are connected to form the matrix.

To illustrate how the Digital Clay crust can be used for modeling, we present a model of an automotive front end. The designer can manipulate 12 independent inputs to control various aspects of the front end's shape as seen in Figure 4.3. A formable crust design is used to model the hood.



Figure 4.3: Low Degree-of-Freedom Car Hood Model

The crust is manually actuated using 1 DoF levers. The levers amplify the inputted user displacement and control the crust surface through the beams shown in Figure 4.4 and 4.6. This allows the flat surface of the crust to morph into various car hood designs. For clarity, not all levers and compliant mechanisms are shown in the Figures. A formable crust is attached to the beam and column tops. As one can see, even though only one unit cell is actuated, the neighboring cells will deform as well.

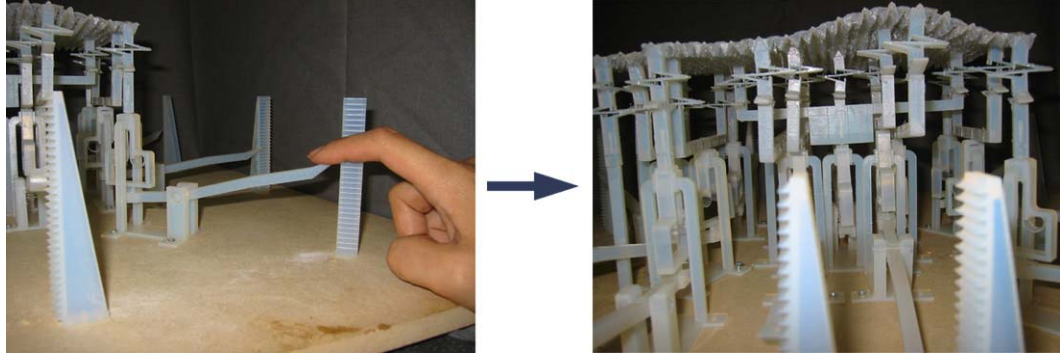


Figure 4.4: Deformation of Car Hood Model

The user can control the hood shape by manipulating the positions of points on the hood. The motions of many of these points are coupled due to the rigid linking of the unit cells. The objective is for the crust to start as a flat surface that can morph into various hood designs of high-end sports cars, such as the Lotus, Ferrari, and Corvette, sketches of which are shown in Figure 4.5. The morphing of the crust can be controlled by changing the points that are actuated.

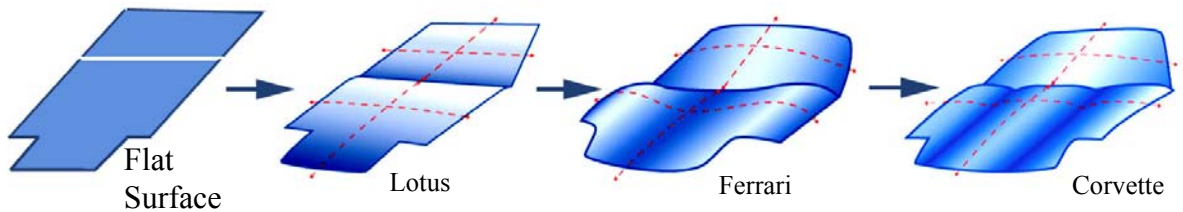


Figure 4.5: Morphing of the Car Hoods

To connect the inputs to the crust, a set of compliant mechanisms was integrated with the beams that are connected to the crust. The arrangement of columns and beams to control positions of points on the crust is shown in Figure 4.6.

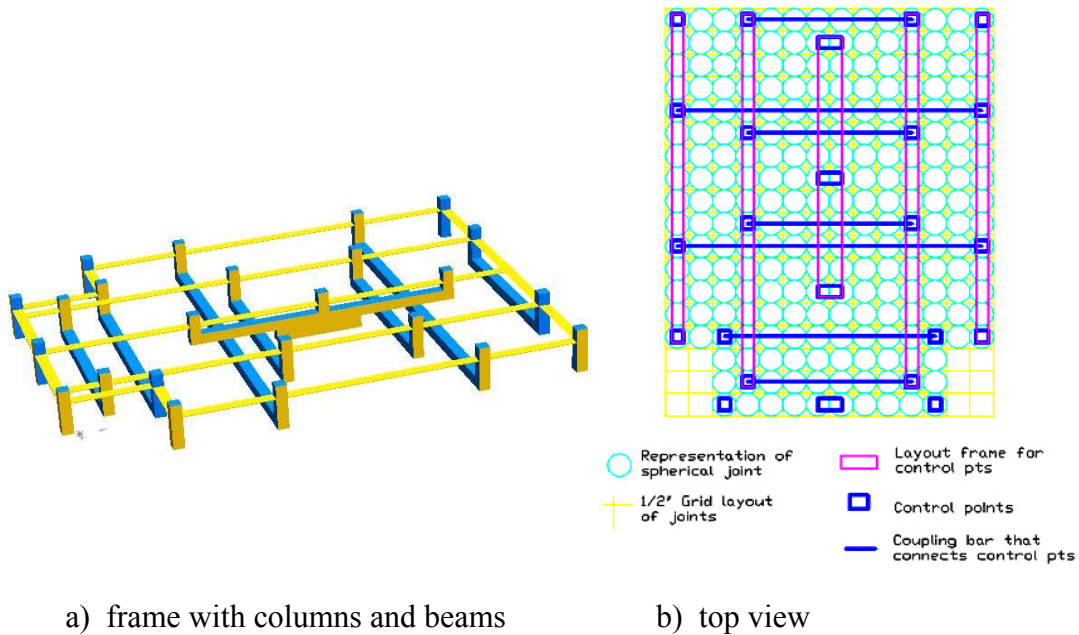


Figure 4.6: Car Hood Frame

The two front corner columns are fixed, while the other columns can be vertically displaced and flexed laterally when necessary to produce smooth surfaces. The beams ensure longitudinal symmetry of the hood, which can be seen in Figure 4.6b. Each beam will be driven vertically by one compliant mechanism. The two columns at the top of the hood (hood-windshield joint) are coupled by another beam (not shown). The crust consists of a 14-by-18 array of spherical joints (Figure 4.6b) that are spaced 12.7 mm apart to give an overall size of about 177.8 –by– 228.6 mm (7 X 9 inches).

Covering the crust will be a flexible skin. The actuators and columns will be rigidly attached to a base. We have considered adding a windshield and fenders (non-formable) to complete the model, but at this stage the hood will be enough to demonstrate the deformation of the crust.

The car hood model is only one example of how the deformable crust matrix can be used. There are various other applications for the Digital Clay crust matrix. The question is, which points or unit cells of the crust matrix need to be actuated and how much control is necessary to create the various desired surfaces (e.g., car-hood models)? This chapter will discuss two methods for predicting the deformation of the crust. Both methods form their prediction using the system constraints, material properties, and magnitude and location of the points actuated.

For both methods, the inputs to the system are the Z coordinates of the centers from various unit cells in the matrix. Figure 4.7 shows an example of how the Z -height input can affect the links. Since the link is rigid, it cannot stretch as seen in the figure.

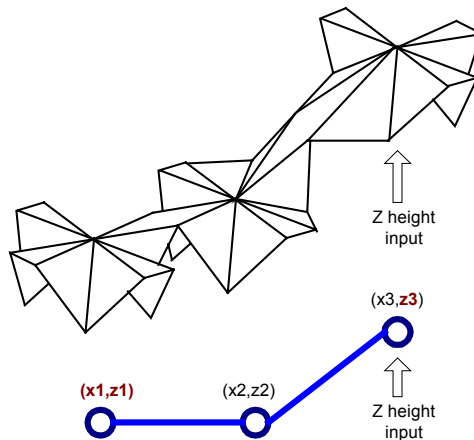


Figure 4.7: 2D Example of Input

In addition, because of the coupling between each cell and its neighbors, it is not necessary for all the Z -heights to be inputted for the program to find all of the coordinates of the center-points for every unit cell. Even if all the Z -heights are known, there is still

not have enough equations to fully describe the surface. For every Z-values there are two unknowns-- X and Y values. Based on this fact, the system is under-constrained. To solve this problem, the two methods will use an iterative process that searches through various combinations of cartesian coordinates for the lowest possible potential energy of the system while maintaining the constant length between two unit cells. The basic structure of both methods is as explained in Figure 4.8.

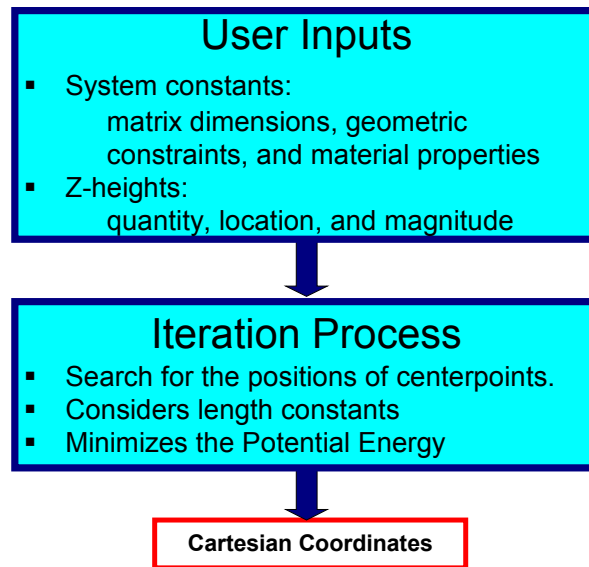


Figure 4.8: Basic Structure of Both Methods

The first method uses an abstract model of the matrix of unit cells as seen in Figure 4.9. The balls represent the center points of the unit cells in the matrix. The lines that connect the balls represent the rigid links between the cells. As the surface deforms the balls move and the ends of the links are free to pivot about the balls. Rotational springs are used to model the stiffness of the compliant joints in the unit cells. The spring constant is an average stiffness value of each compliant joint.

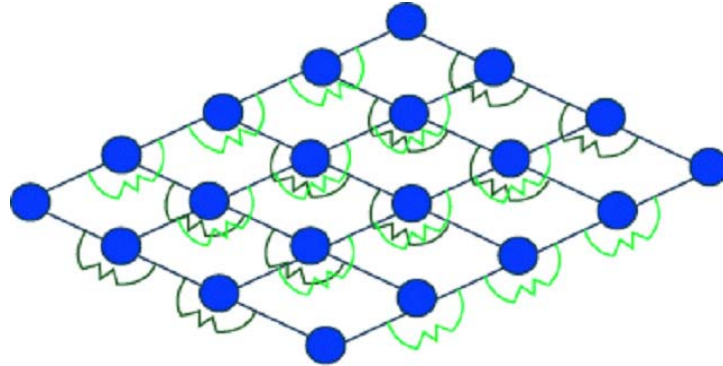


Figure 4.9: Method 1-- Abstract Model

The second method is a more accurate representation of the crust model that better represents the manufacturable shape and behavior of the formable crust. The model uses two different stiffness values to represent the two different joint designs for each unit cell. Figure 4.10 shows where all the springs are located for one unit cell in the model used by the second method. Joints 1 and 3 have the same stiffness value while Joint 2 has a different value.

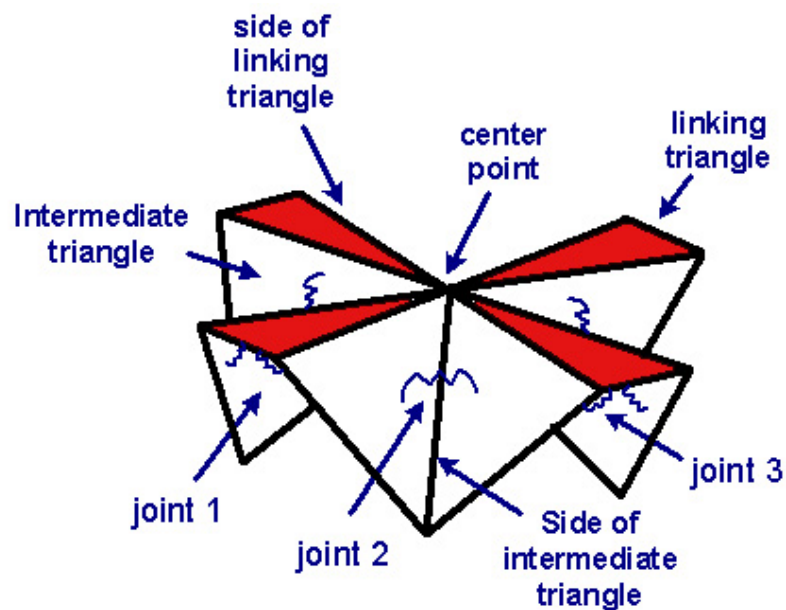


Figure 4.10: Springs for One Unit Cell

Now that both two methods are introduced for predicting the shape deformation of the crust matrix based upon several parameters, the math will be explained. The results will be the cartesian coordinates (x,y,z) for the centers of every unit cell in the matrix. In the following sections, the first set of sections will be discussing principles and pseudo-code for the problem formulations, flow charts, and algorithms for the Method 1. This will be called Part 1: Method 1. The second set will be discussing about Method 2. This will be called Part 2: Method 2. Then the third will be about the forward and inverse statics. Following the third set is one section discussing about the number of unknowns, number of equations, and Degrees-of-Freedom for both methods.

PART 1 METHOD 1: ABSTRACT FORMABLE CRUST MODEL

Below is a flow chart for the first method. The numbering on the right side denotes the section that will explain the block. Following the flow chart is a step-by-step description of the algorithm used in the first method.

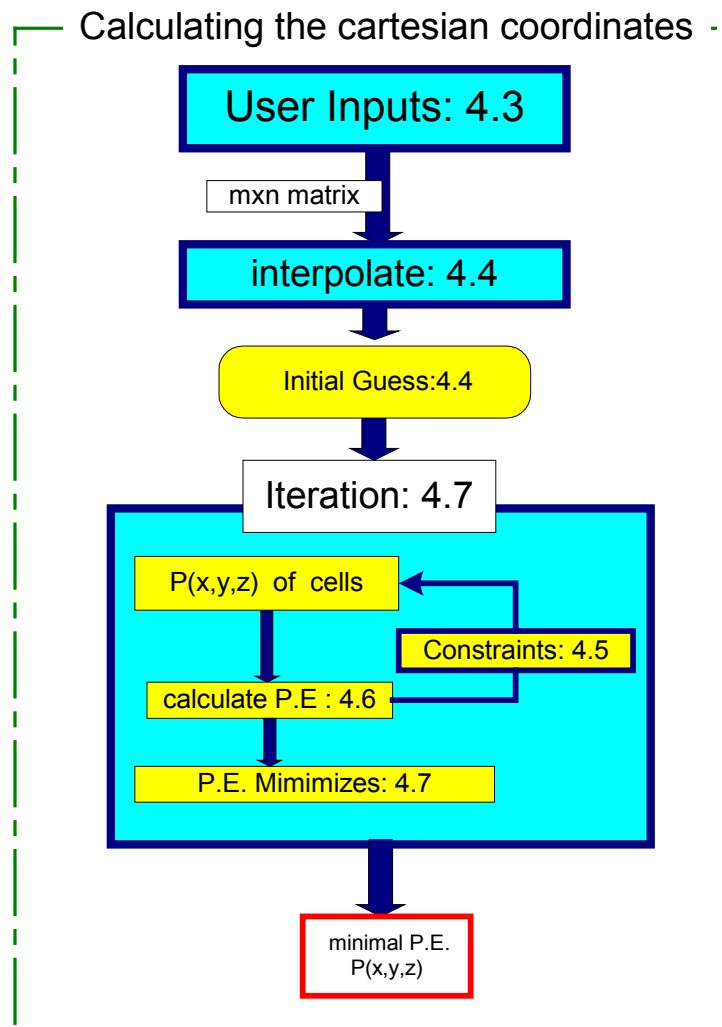


Figure 4.11: Flow Chart of Method 1

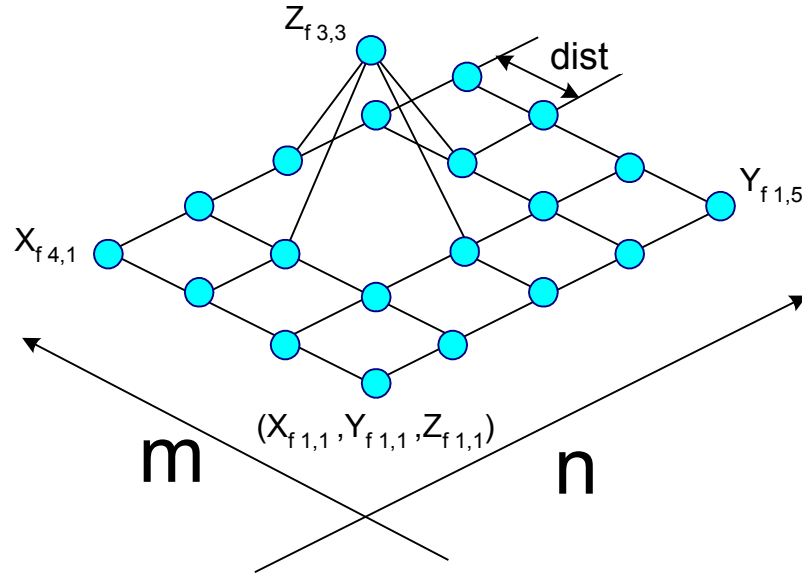


Figure 4.12: Example of matrix initial conditions

Note that only the $P_{1,1}$ corner point has fixed X, Y, and Z values.

Given:

m -by- n = size of matrix of unit cells center points in a uniform 2D grid

$Z_{f_{i,j}}$ = fixed Z value of the center-points for the $[i,j]$ cell

$X_{f_{i,j}}$ = fixed X value of the center-points for the $[i,j]$ cell

$Y_{f_{i,j}}$ = fixed Y value of the center-points for the $[i,j]$ cell

dist = distance between two center points

k = average stiffness value of all the joints in the matrix

Find: $P_i(x,y,z)$ = center-points of every unit cell in the matrix

Satisfy: $\|P_i - P_{i+1}\| = \text{dist}$ (Equation 4.1)

where P_i and P_{i+1} are adjacent unit cell centers and Equation 4.1 holds for every pair of adjacent cells.

Minimize: $P.E = \sum_{i=1}^{m*n} \frac{1}{2} * k(\theta_{id} - \theta_{ir})^2$ (Equation 4.2)

where θ_{id} and θ_{ir} are the angles between two center-points with $d =$ after deformation of the matrix deformation and $r =$ rest or before deformation.

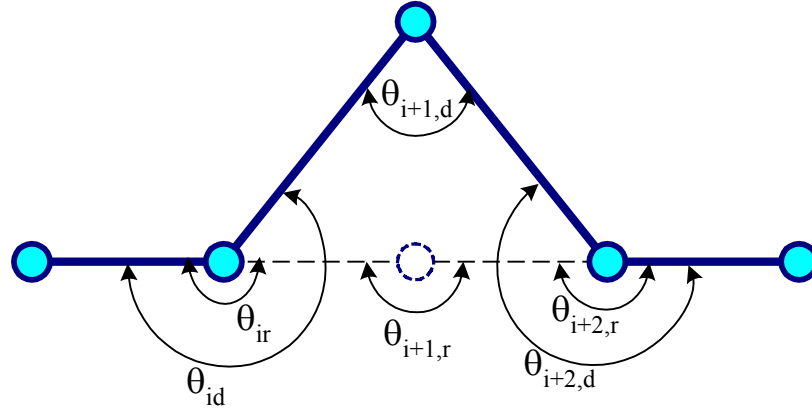


Figure 4.13: the Angles Between Two Center-points

Algorithm:

- 1) Initialize an m -by- n matrix with the fixed cells at Z_{fij} , X_{fij} , Y_{fij} and all other cells at their natural, un-deformed location. Figure 4.12 shows an example of an initial configuration of a matrix.
- 2) Create the initial guess vector of unknowns: x_0
 - a) Linearly interpolate between the fixed Z_{fij} values.
 - b) Place all coordinates into the x_0 vector

- 3) Iterate to find minimum energy combination of the cartesian coordinates
 - a) Minimize total P.E. (Potential Energy) while also satisfying the length constraints and the $Z_{f\ i,j}$, $X_{f\ i,j}$, $Y_{f\ i,j}$ constraints.
 - (1) Calculate P.E. of the angles between all adjacent P_i and P_{i+1} .
 - (2) Compare P.E. values from previous results.
 - (3) Calculate length between all adjacent P_i and P_{i+1} to check the length constraint.
 - (4) Check the $Z_{f\ i,j}$, $X_{f\ i,j}$, $Y_{f\ i,j}$ constraints.
 - (5) Modify guesses
- 4) Output the $P_i(x,y,z)$ of all the unit cells

4.1 Referencing Notation

Before presenting the math, the notation for referencing cells in the crust matrix will be discussed. Figure 4.14 shows a matrix of balls that represent the center points of each unit cells. Figure 4.14 also demonstrates how each cell in the matrix is referenced. One way is through cartesian coordinates of their center-points. A second way is by their chronological sequence in the matrix (e.g. P_{14}). A third way is to reference to the rows and column location of each cell: $P_{i,j}$, where i and j denotes the row and column respectively. For example, P_{14} means the same thing as $P_{3,4}$. The type of referencing used will change based upon convenience and what context it is used in.

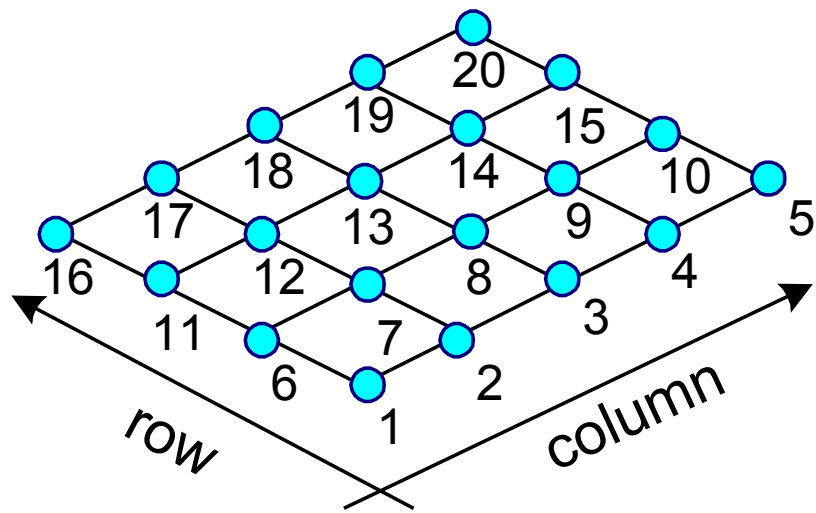


Figure 4.14: Counting Convention

4.2 The Principles Underlying the Formable Crust Models

The two methods mentioned earlier have similarities in how they approach the problem. First we will discuss the mathematical principles behind Method 1. Method 2 will follow Method 1. The organization of the principles is based upon the previous mentioned flow chart: Figure 4.11.

4.3 Constraints From User Inputs (Method 1: “User Inputs” Block)

For all cases, the first cell, P_1 , will always be fixed at $(0,0,Z_{f1,1})$. The $Z_{f1,1}$ is either zero or any value that the user inputs. At every corner including the first cell, all the z values are also constrained at either ‘0’ or any value specified by the user. For example, in Figure 4.15 the user has specified several Z -heights at P_{13} and P_5 , which become additional constraints. Also the upper left X - coordinate, P_{16} in Figure 4.15, and the lower right Y - coordinate, P_5 , are fixed at zero or at the user specified value.

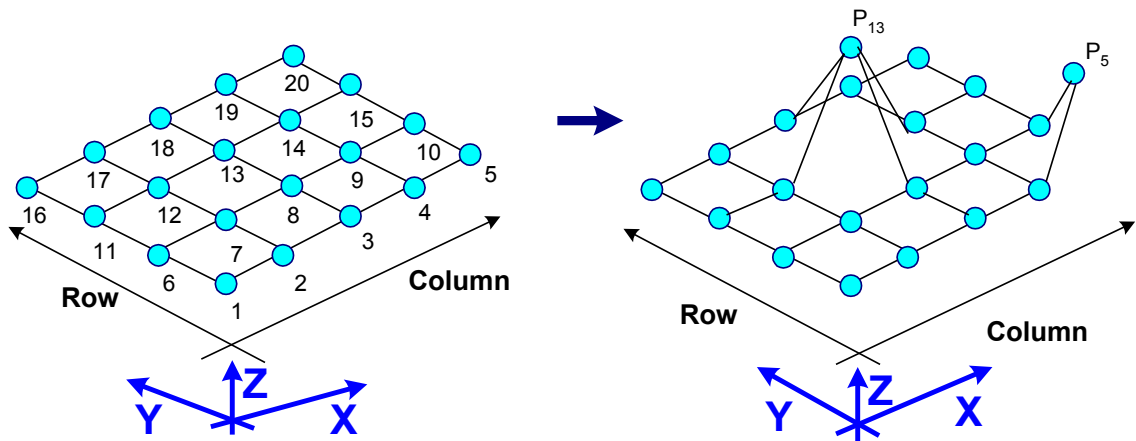


Figure 4.15: Inputting Z-heights

4.4 Line Interpolation (Method 1: “Interpolate” block)

Once the constraints are added, the balls, representing the center-points of the unit cells, are moved to create a linear interpolation between two constraining coordinates. This interpolation becomes the initial guess of the deformed state for the minimization program. Because it is difficult to show the interpolation for a 3D matrix, the 2D example in Figure 4.16 will demonstrate how the linear interpolation works.

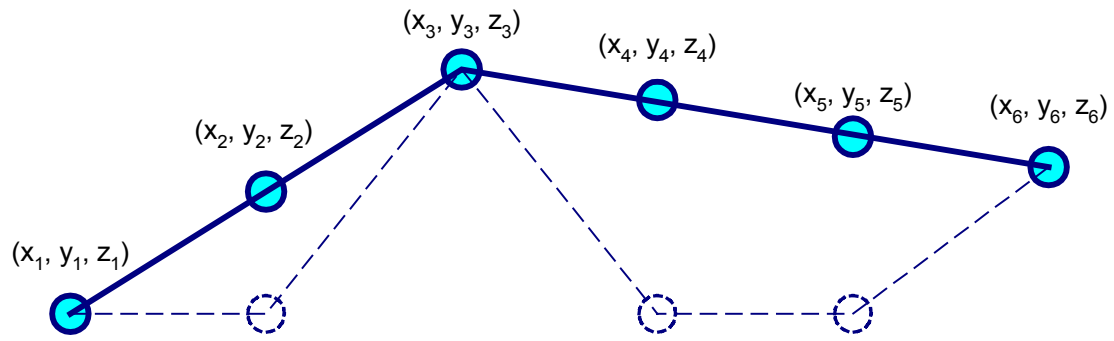


Figure 4.16: Line interpolation

Linear interpolation of the unknown variables improves the initial guess of the unknowns. In most numerical methods, which use iterative methods to find a minimum or maximum of a system, a good question to ask is: is the result a global or a local minimum? By using linear interpolation, the initial guess is placed closer to the global minimum. Therefore the program will converge faster and be more likely to converge toward a global minimum.

4.5 Constraints (Method 1: “Iteration” Block)

Constraints are equations that limit the iteration process from deviating far from the correct answer. Below are two types of constraints.

4.5.1 Length Constraints

Since each cell is rigidly linked, the distance between any two unit cells is constant. Therefore, the calculated length in either the x or the y direction minus the user specified constant distance, “dist”, should equal a number that is either zero or close to it:

1) Length constraint in the x-direction

$$a) \quad L_x = (X_{i,j} - X_{i+1,j})^2 + (Y_{i,j} - Y_{i+1,j})^2 + (Z_{i,j} - Z_{i+1,j})^2 - \text{dist}^2 \approx 0 \quad (\text{Equation 4.3})$$

2) Length constraint in the y-direction

$$a. \quad L_y = (X_{i,j} - X_{i,j+1})^2 + (Y_{i,j} - Y_{i,j+1})^2 + (Z_{i,j} - Z_{i,j+1})^2 - \text{dist}^2 \approx 0 \quad (\text{Equation 4.4})$$

With ‘i’ representing the rows and ‘j’ representing the columns

Dist= the inputted distance value between two unit cells

X, Y= cartesian coordinates for the unit cells

4.5.2 Coordinate Constraints

Based upon the user inputs and the embedded constraints, several of the coordinates are constrained. Below are equations that constrain the coordinates. Any variable with the subscript “f” denotes a fixed coordinate value that should be constrained. The variable without the subscript is the one that the iteration process calculated:

X-Coordinate Constraints

$$X_{fi,j} - X_{i,j} = 0 \quad (\text{Equation 4.5})$$

Y-Coordinate Constraints

$$Y_{fi,j} - Y_{i,j} = 0 \quad (\text{Equation 4.6})$$

Z-Coordinate Constraints

$$Z_{fi,j} - Z_{i,j} = 0 \quad (\text{Equation 4.7})$$

4.6 Calculating Energy (Method 1: “Iteration” Block)

The deformation of the crust matrix is predicted by minimizing its potential energy. As previously mentioned in the introduction of this chapter, the first method models the matrix as a system of springs, rods, and balls. These rotational springs connect two rigid links together, causing the links to become pivoting rods. The stiffness value is the average of the stiffness value in the system. The equation for calculating the energy is already mentioned in Equation 4.2 and will not be repeated here.

To calculate the energy, several variables need to be known. They are the stiffness value, the angles, and the positions of the center-points of the unit cells. They are derived in the following sections.

4.6.1 Finding the Stiffness Value

From the previous section, calculating the potential energy requires knowing the stiffness value, k . For one unit cell there are two k values because there are two different joint designs as seen in Figure 4.17.

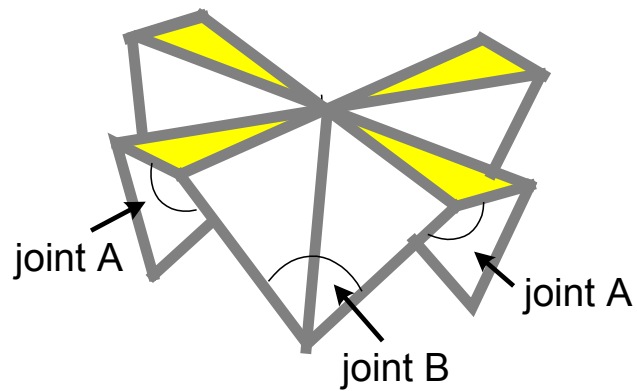


Figure 4.17: The Two Different Joint Designs

To study each stiffness value, the joints are “cut-up” or individually modeled and manufactured in the stereolithography material that is being used for manufacturing the crust. The k values are then found through performing several experiments.



Figure 4.18: Joint A (Left) and Joint B (Right)

4.6.1.1 The Experimental Set-up

The set-up uses the Instron Universal Material Testing Machine as seen in Figure 4.19 with the three images.

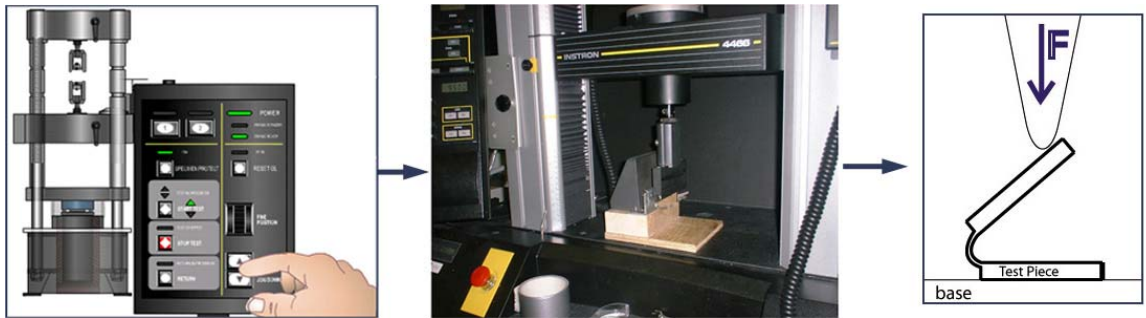


Figure 4.19: Experimental set-up for Finding Stiffness

The first image is the Instron Testing Machine. The second image is a zoom-up of the attachment piece that applies the force to the specimen being tested. The Instron is also connected to a computer that reads the force versus the displacement of the specimen as the force is being applied. The last image is a diagram of how the force is being applied to one of the specimens while the specimen is being attached to a base. The maximum displacement is 10mm from the starting position.

4.6.1.2 Joint A (Larger Joint) Design and Results

As previously mentioned, the joints are individually modeled so that they can be tested independent of the other joints. The dimensions for joint A are seen on the left image in Figure 4.20. The free-body diagram is on the right of joint A as it is being set-up and tested. For this case, the test is performed vertically where the specimen is attached to the sidewall of the base.

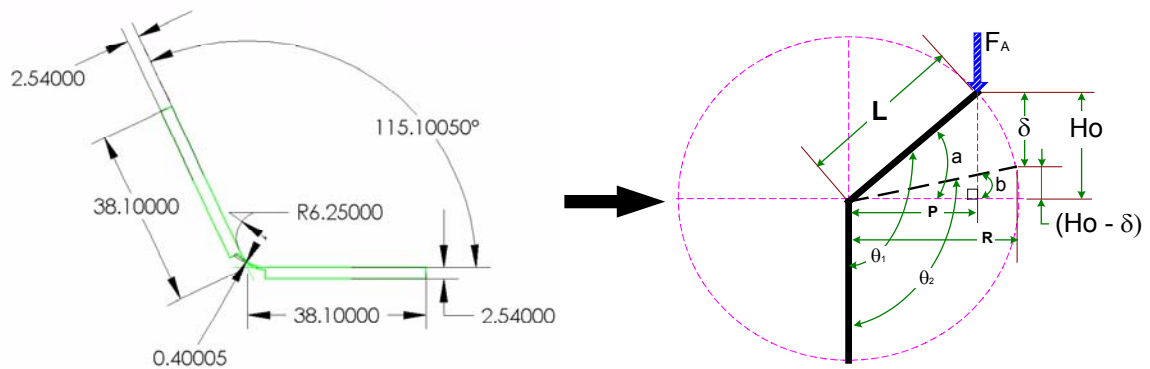


Figure 4.20: Set-up for Joint A

The mathematics is as follows for finding the torque based upon the force versus displacement values. From the torque, we can find the stiffness value for joint A.

Definition: L =Length of the axial arm

δ (mm) = The displacement value/s as the force is being applied to the specimen

F_A (N) = The force that is being applied to the specimen

θ_1 = The original angle between two arms before deformation

θ_2 = The rotational angle between the arms

H_o = Beginning height of the axial arm where the force is being applied

P = The horizontal length from where the force is being applied (Constant Value)

R = The responding horizontal length that changes with the force

Angle a = Original angle from axial arm to horizontal plane before deformation

Angle b = rotational angle of axial arm measured from horizontal plane.

Given: $L = 38.1\text{mm}$, δ , F_A , θ_1

Find: $T = \text{Torque}$

Equations:

$$H_o = L \sin(a) \quad (\text{Equation 4.8})$$

$$b = \sin^{-1} \left(\frac{H_o - \delta}{L} \right) \quad (\text{Equation 4.9})$$

$$\theta_2 = 90 + b \quad (\text{Equation 4.10})$$

$$\text{Torque} = P \times F_A \quad (\text{Equation 4.11})$$

From the equations and the data, we can find the torque. From the torque we can find the stiffness based upon the slope of the torque graph. Figure 4.21 is an example graph for the torque and the linear fitting line for the torque.

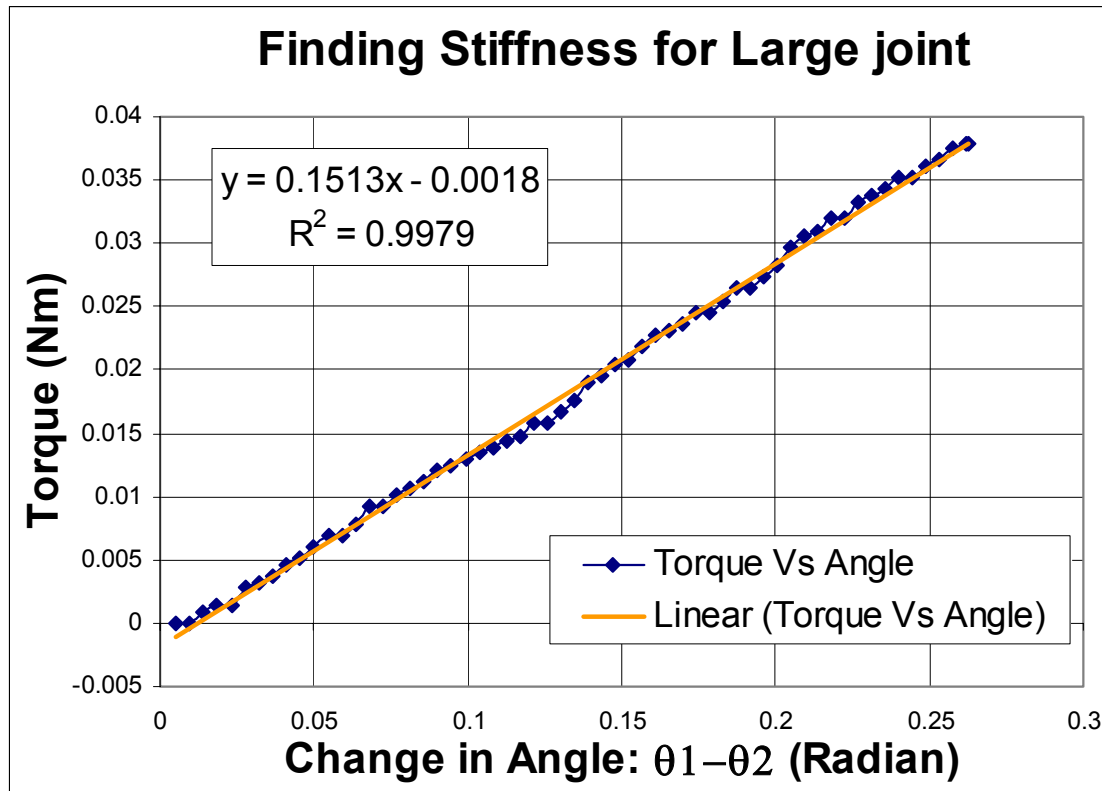


Figure 4.21: Finding Stiffness for the Joint A—the Larger Joint

From Figure 4.21, the fitting equation is $y = 0.1513x - 0.0018$ with $R^2 = 0.9979$. This means that stiffness value is 0.1513 Nm with the y-intercept of -0.0018 . After 9 experimental runs, the average stiffness value is **about 0.1348 Nm**. (See Appendix A)

4.6.1.3 Joint B (Smaller Joint) Design and Results

Similar to Joint A, the dimensions and the free-body diagram are shown below.

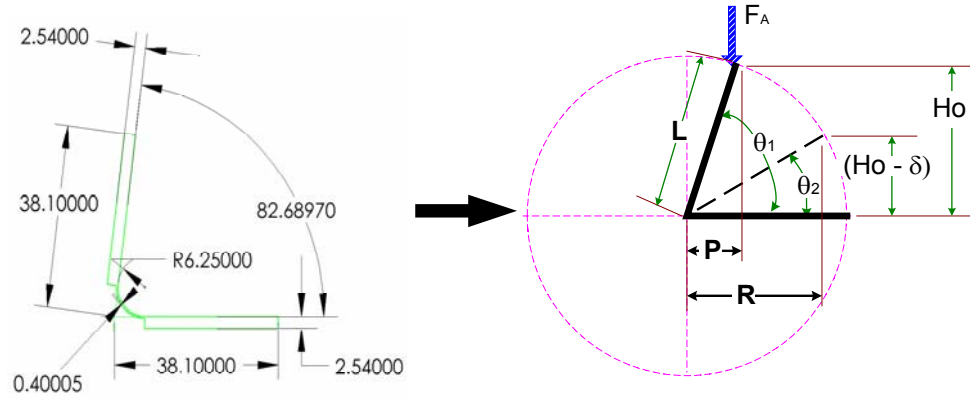


Figure 4.22: Set-up for Joint B

For this case, the joint is attached horizontally on top of the base and the force is applied as shown. Again we will find torque first and then find the stiffness.

Given: $L = 38.1\text{mm}$, δ , F_A , θ_1 , H_o

Find: $T = \text{Torque}$

Equations:

$$H_o = L \cdot \sin(\theta_1) \quad (\text{Equation 4.13})$$

$$\theta_2 = \sin^{-1}\left(\frac{H_o - \delta}{L}\right) \quad (\text{Equation 4.14})$$

$$\text{Torque} = P \times F_A \quad (\text{Equation 4.16})$$

Similar to joint A, the torque is found from the equations developed above and the experimental data as seen in Figure 4.23. From the torque, the stiffness can be found by fitting a trend line through the points.

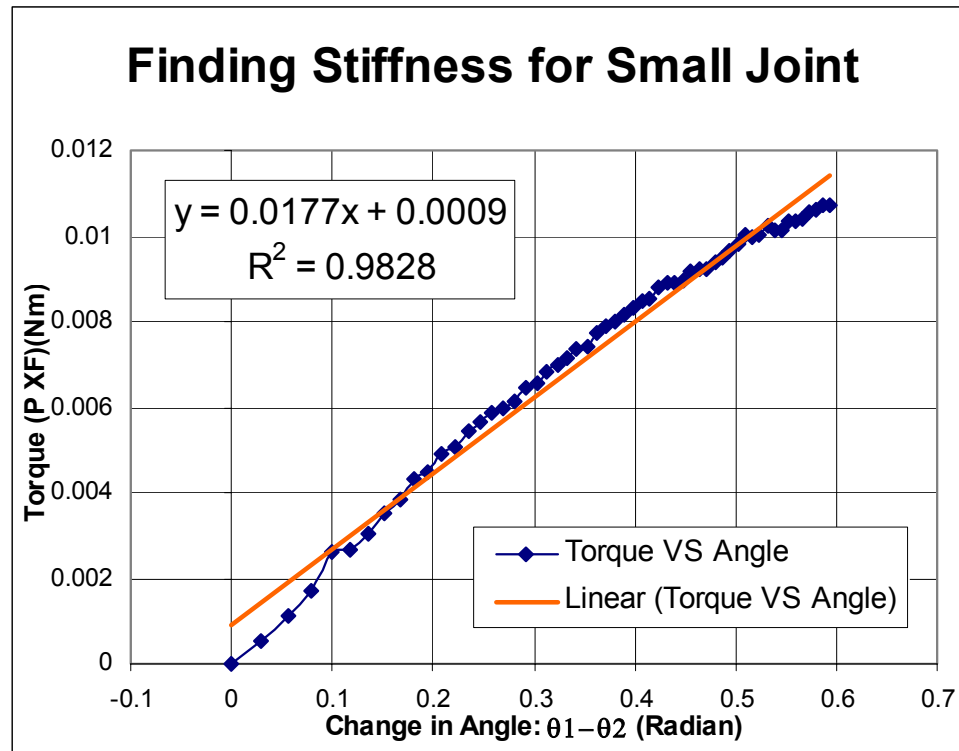


Figure 4.23: Finding Stiffness for Joint B (smaller Joint)

The linear fitting line shows that the stiffness value for this particular specimen is 0.0177 with the R^2 value of 0.9828. After 6 experimental runs, the average **stiffness value** for 4 of the experimental runs **for joint B** is **0.01635 Nm**. Two of the runs were outliers and were thrown away. (See Appendix A)

4.6.1.4 Effect of k

After finding the k values, how does the value of the k's affect the results? The effect of k can be determined analytically and numerically and be verified by performing various runs of the two methods. For Method 1, the value of k does NOT have a significant effect on resulting deformations. For finding a local minimum of any function, the derivative of the function needs to be zero. For this problem we are searching for minimum energy stored in the system. When taking the derivative of the potential energy equation with respect to the variables that are being minimize (eg. X_i, Y_i , and Z_i of the center-points) for Method 1, the constant k value can drop out as seen in Equation 4.17. This makes sense because the constant k value is a scaling factor that does not affect the location of the minima of the energy function. As a reminder, the X_i, Y_i , and Z_i of the center-points are embedded in the θ_i coefficients. These variables determine the θ_i values as shown in earlier sections.

$$\frac{\partial \left(\frac{1}{2} * k * \sum (\theta_{id} - \theta_{ir})^2 \right)}{\partial x_i} = 0 \quad \text{(Equation 4.17)}$$

After explaining the analytical reasoning, now we will discuss the effect of the k value on the numerical algorithm for finding the minimum of the problem. For Method 1 the k value is an amplitude that can be moved out of the summation term as shown in Equation 4.17. Similarly for matrices such as the Jacobian or the Hessian, the k value can be moved out of the matrix. The Matlab algorithm uses the Jacobian and the Hessian matrices to find the direction of the search vector to where the minimum is thought to lie.

The k will act like a magnification that lengthens the search vector direction. It does not improve the speed of convergence and does not change where the minimum is thought to lie. If the $k=0$, then there isn't a search vector direction for the algorithm to search for. Setting $k=0$ will create an error in the algorithm. The sum of potential energy will be amplified by the value of k . With different k values, the same minimum point can be found except the total energy at that point will be different. Of course, the specific values of k will become more important if nonlinear effects are taken into account.

Figure 4.24 shows the different results with different k values for a 4-by-5 matrix with two inputs.

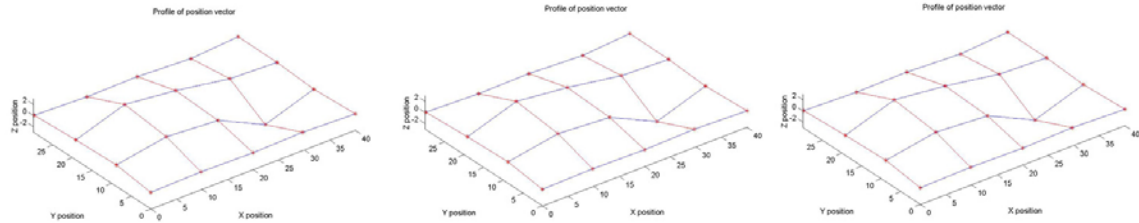


Figure 4.24: 4-by-5 Method 1 K Test. $K=1$ (1st image). $K=100$ (3rd). $K=1000$ (4th)

If the values of the results from Figure 4.24 are compared, their differences are negligible.

Since the k value does not matter, is it necessary to use energy minimization to find the position of the center-points based upon a handful of inputs? Answer: No. This is an under-constrained system with several fixed values. Within the system are length constraints that restrict the distance between any two unit cells. The purpose is to find all

the coordinate locations of every center-point of the unit cells. One possible solution without using a system of springs is applying a surface-fitting algorithm to fit to the input X, Y, and Z coordinates. The surface-fitting algorithm could be to a least squares regression algorithm for interpolating polynomial surfaces of various orders. The greatest challenge is applying the length constraint which makes the algorithm not a direct process but an iterative guessing process until the length constraints are satisfy within reasonable error values.. However it is beyond the scope of this thesis to further develop this idea.

For Method 2, there are two k values. These values do have an effect on the deformation as shown in Equation 4.18.

$$\frac{\partial \left[\frac{1}{2} * \sum \left[\frac{k_1}{k_2} [(\theta_{1p} - \theta'_{1p})^2 + (\theta_{3p} - \theta'_{3p})^2] + (\theta_{2p} - \theta'_{2p})^2 \right] \right]}{\partial x_i} = 0 \quad (\text{Equation 4.18})$$

As one can see, the Method 2 energy equation is re-arranged by dividing the first summation by the k_2 value; then taking the derivative with respect to a X_i value. For this case the $\frac{k_1}{k_2}$ cannot be removed because it is only part of one term in the summation. As a reminder, k_3 from the energy equation (not shown in Equation 4.18) is an arbitrary large number that would guide the iteration process from deviating from the given constraints. Like Method 1, k_3 can be moved out of its summation term and be removed. It does not guide the convergence as much as the other k's would. However it will affect the total potential energy value as explained in previous sections. Figure 4.25 shows the different

results from the different combination of k values using the same coordinate inputs and constraints. Again the units are not important if all the inputs use the same units.

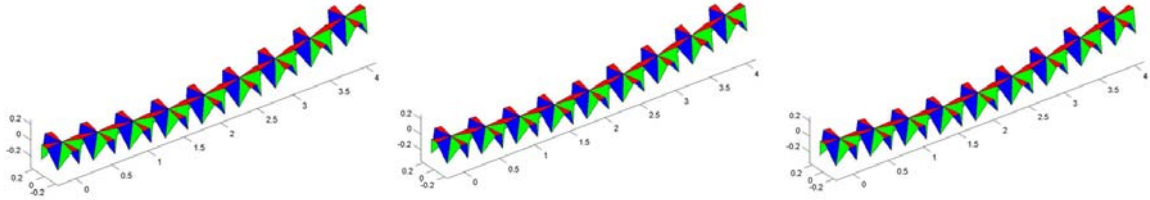


Figure 4.25: $K_1=100$; $K_2=500$ (LT). $K_1=500$; $K_2=500$ (M). $K_1=100$; $K_2=1000$ (RT)

With just visual inspection, all the graphs look the same. By comparing the actual coordinate values, there is a $1/100$ difference in the coordinate values. However, the k_1 and k_2 values are not as important as the ratio of these two values as shown in Equation 4.18. For an example, $k_1=10$ and $k_2=100$. The ratio of the two k values is $10/100 = 1/10$. That means that as long as the ratio of the k values is maintained, different runs with different k values with the same constraints will have similar (if not the same) deformation results. Refer again to Figure 4.25 for examples except change the values of the k s to $k_1=100$ $k_2=500$ (Left) $k_1=10$ $k_2=50$ (Mid) $k_1=1$ $k_2=5$ (Right). The resulting coordinate values do change, but by the factor of $1/1000$. The difference is less than the previous example.

4.6.1.5 Stiffness Ending Remarks

After finding both stiffness values for the two joints in the unit cell, the equation for calculating the potential energy for the system is complete. The first method uses a simplified model, with only one spring per unit cell. Therefore, we will use the average

stiffness value from both joints in method 1. The **average value is 0.0756 Nm**. However, the second method uses both stiffness values and is more complex but more accurate.

4.6.2 Calculating the Angles from the Position Coordinates

The angles for Method 1 are the angles between two rigid links as seen in the previous images in Figure 4.9 and 4.13. As they deform, not all the angles stay 180 degree or π . Since the rigid links can be seen as vectors, we will apply ATAN2 to calculate the angles between any two vectors. For a matrix of any size, the angles are arbitrarily calculated first in the x-direction then in the y-direction. Since the deformations are in the Z-axis direction, the calculation will use the z values from each position coordinate. The $-\pi$ value insures that the angle we are measuring is below the surface of the matrix.

Angles in the X-directions

$$X_{i,j} = -\pi + \text{atan2}\left[Z_{i,j} - Z_{i-1,j}, X_{i,j} - X_{i-1,j}\right] - \text{atan2}\left[Z_{i-1,j} - Z_{i-2,j}, X_{i-1,j} - X_{i-2,j}\right] \quad (\text{Equation 4.19})$$

Angles in the Y-directions

$$Y_{i,j} = -\pi + \text{atan2}\left[Z_{i,j} - Z_{i,j-1}, Y_{i,j} - Y_{i,j-1}\right] - \text{atan2}\left[Z_{i,j-1} - Z_{i,j-2}, Y_{i,j-1} - Y_{i,j-2}\right] \quad (\text{Equation 4.20})$$

4.7 Iteration Process (Method 1: “Iteration” Block)

The iteration process applies a “search and calculate” process. The search for a minimum energy position of the unit cells starts by following a gradient path. Then the

potential energy of the matrix is calculated at the end of the path. Next, this value is compared to the previously calculated potential energy at the end of the previous path. One successful iterative process that we found is the “Fmincon” process. “Fmincon” is a pre-packaged MATLAB minimization program that searches for the minimum of a constrained nonlinear multivariable function by computing a sparse finite-difference approximation to the Hessian Matrix (Mathworks, 2004). For this case, “Fmincon” will search through various combinations of cartesian coordinates to find the minimum potential energy of the system based upon the initial guess. Second if “Fmincon” is supplied a series of nonlinear constraints, such as the constant length of the unit cells, the Hessian matrix will be restricted by using another matrix of equations while searching for feasible answers (Gill, et. al, 1981). ‘Fmincon’ is best described as two steps: ‘Determination of a Direction’ and ‘Line Search Procedure’. Below is a simple overall summary of the math. It is beyond the scope of this project to decipher the math behind these two steps.

4.7.1 Determination of a Direction using the Hessian Matrix

The Hessian matrix contains the second partial derivatives of a function at various unknown variables. For this case the function is the Potential Energy equation with the unknowns coordinate values. The Equation 4.21 describes the Hessian matrix. The matrix be positive definite to insure that the line is going in the right direction. “Hessian, H , is always maintained to be positive definite so that the direction of search, d , is always in a descent direction” (Mathworks). For every small step in the direction, d , the potential energy function will decrease in magnitude.

$$\text{Hessian} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \dots \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \dots \\ \vdots & \vdots & \ddots \end{bmatrix} = + \textit{definite} \quad (\text{Equation 4.21})$$

4.7.2 Line-Search Procedure

‘Line-Search Procedure’ is done by following along the line created by the Hessian matrix and searching for the location that is at the lowest value.

4.8 Ending Comments for Method 1

Using the principles mentioned above, the minimal cartesian coordinate positions for every unit cell in the matrix of any size can be calculated. The joint angles can also be calculated after determining the cartesian coordinate position by applying the joint calculation process that will later be explain in Part 2: Method 2 of this chapter.

PART 2 METHOD TWO: ACTUAL MANUFACTURABLE CRUST MODEL

Unlike Method 1, which uses an abstract model of the crust, Method 2 computes the shape of the actual manufacturable, formable crust. Inverse and forward statics and spherical coordinates matrix manipulations are used to find all the unit cell positions and the crust shapes. Below is the flow chart with S.C and C.C standing for spherical and cartesian coordinates respectively. All other notation has already been defined in previous sections. Following the chart is the step-by-step algorithm formulation for Method 2.

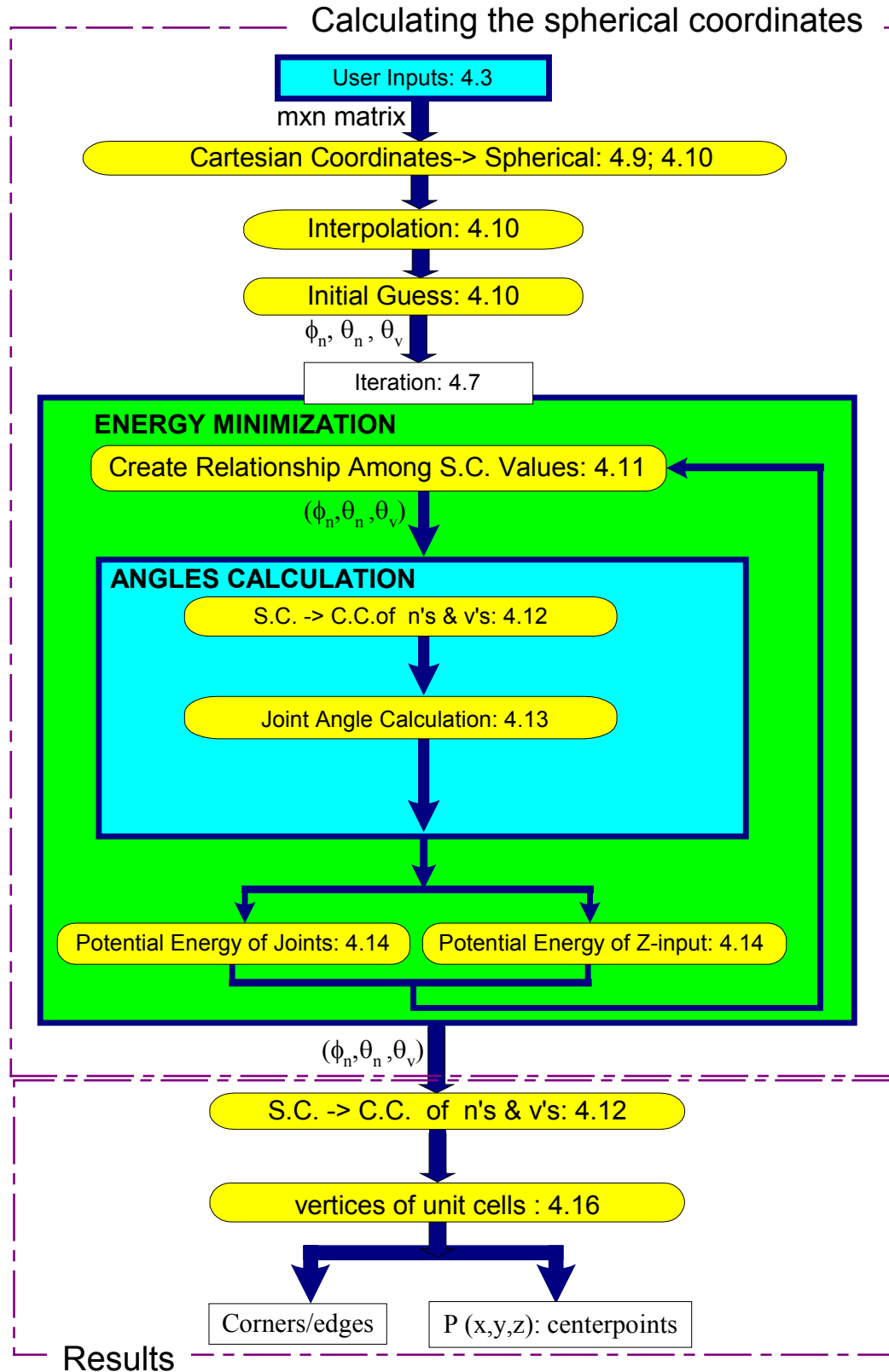


Figure 4.26: Flow Chart of Method 2

Given: m-by-n, Z_{fij} , X_{fij} , Y_{fij} , dist, L_1 , L_3 , La , α_1 , α_2 , β_1 , β_2 , k_1 , k_2 , k_3 , θ_{1p} , θ_{2p} , θ_{3p}

New Given variables not previously explained:

k_1 = stiffness value of Joints 1 and 3 (see Figure 4.10)

k_2 =Stiffness value of Joint 2 (see Figure 4.10)

k_3 = arbitrarily large stiffness value to control the Z-heights

$\theta_{1p}, \theta_{2p}, \theta_{3p}$ = Joint angles of the structure in its undeformed state. θ_{1p}

refers to the angle at Joint 1 in Figure 4.10; θ_{2p} refers to the angle

at Joint 2 in Figure 4.10; and θ_{3p} refers to the angle at Joint 3 in

Figure 4.10. Note that this set of 3 angles repeats itself 4 times in

each unit cell. Therefore index “p” runs from 1 to $4*m*n$ where m

is the number of rows and n is the number of columns in the matrix.

$L_1, L_3, La, \alpha_1, \alpha_2, \beta_1, \beta_2$ =Geometrical constants for the 3-D structure.

These will be explained later.

Find: $P_i(x,y,z)$ = center-points of every unit cell in the matrix

Satisfy: Z_{fij} , X_{fij} , Y_{fij} Where these constraints remain fixed values;

distance constraints

Minimize:

$$\begin{aligned}
P.E = & \sum_{p=1}^{4*m*n} \frac{1}{2} * [k_1(\theta_{1p} - \theta'_{1p})^2 + k_2(\theta_{2p} - \theta'_{2p})^2 + k_1(\theta_{3p} - \theta'_{3p})^2] \\
& + \sum_{i=1}^m \sum_{j=1}^n \frac{1}{2} * k_3 [(X_{f(i,j)} - X'_{f(i,j)})^2 + (Y_{f(i,j)} - Y'_{f(i,j)})^2 + (Z_{f(i,j)} - Z'_{f(i,j)})^2]
\end{aligned}
\tag{Equation 4.22}$$

$\theta'_{1i}, \theta'_{2i}, \theta'_{3i}$ = Joint angles of the structure in its final, deformed state.

Notation is similar to undeformed angles described above.

$X_{f(i,j)}$ ' = Calculated X-height of the center of the unit cell in the i^{th} -row and j^{th} -column

$Y_{f(i,j)}$ ' = Calculated Y-height

$Z_{f(i,j)}$ ' = Calculated Z-height

Algorithm:

- 1) Initialize the m-by-n matrix with the fixed cells at $Z_{f(i,j)}$, $X_{f(i,j)}$, $Y_{f(i,j)}$ and all other cells set at their natural, un-deformed state based upon the user inputs (see Section 4.3)
- 2) Calculate \vec{n} and \vec{v} -vectors in Cartesian Coordinates
(The meaning of these vectors will be given later)
- 3) Convert Cartesian Coordinates (C.C.) => Spherical Coordinates (S.C.)
- 4) Linearly interpolate to develop the initial guess, x_0

- 5) Iterate to find the minimum energy state
 - a) Energy minimization
 - (1) Create Relationship among S.C. values
 - (2) Convert back to C.C.
 - (3) Apply Inverse Statics using $L_1, L_3, L_a, \alpha_1, \alpha_2, \beta_1, \beta_2$
 - (4) Calculate P.E. of the joints
 - (5) Compare P.E. values from previous results
 - (6) Modify x_0 of S.C
 - b) End Loop
- 6) Results from the iteration are the S.C. of \vec{n} and \vec{v} -vectors
- 7) Convert S.C. \Rightarrow C.C.
- 8) Find P(x,y,z) by applying dist, $L_1, L_3, L_a, \alpha_1, \alpha_2, \beta_1, \beta_2$

4.9 Spherical Coordinates (Method 2: Interpolation and Initial Guess)

Unlike Method 1, Method 2 uses spherical coordinate values instead of Cartesian coordinates. Below is the explanation that would lead to understanding about the interpolation process and the reason for the variables in the initial guess matrix.

The formable crust matrix consists of a series of \vec{n} and \vec{v} unit vectors that completely determine the cartesian coordinates of all the centerpoints, links, and joints for every unit cell in the matrix. Figure 4.27 shows all the vectors for a 1-by-3 matrix of unit cells. The detailed image shows that the \vec{n} -vectors are the normal vectors to the triangular faces of the unit cells. The \vec{v} -vectors originate from the center of a unit cell and point toward the centers of all adjacent unit cells. They always lie in the planes of the triangular faces, called linking triangles. Note that the \vec{n} -vectors are always perpendicular to the \vec{v} -vectors. By specifying the elements of these vectors and knowing a few geometric constants, the coordinates of any point on any of the unit cells can be calculated. Both the \vec{n} and \vec{v} -vectors consist of three unknowns (x,y,z). For just 2 unit cells, there are 8 \vec{v} -vectors and 8 \vec{n} -vectors, giving a total of $(8+8)*3= 48$ unknowns. One method to reduce the number of unknowns is to use spherical coordinates.

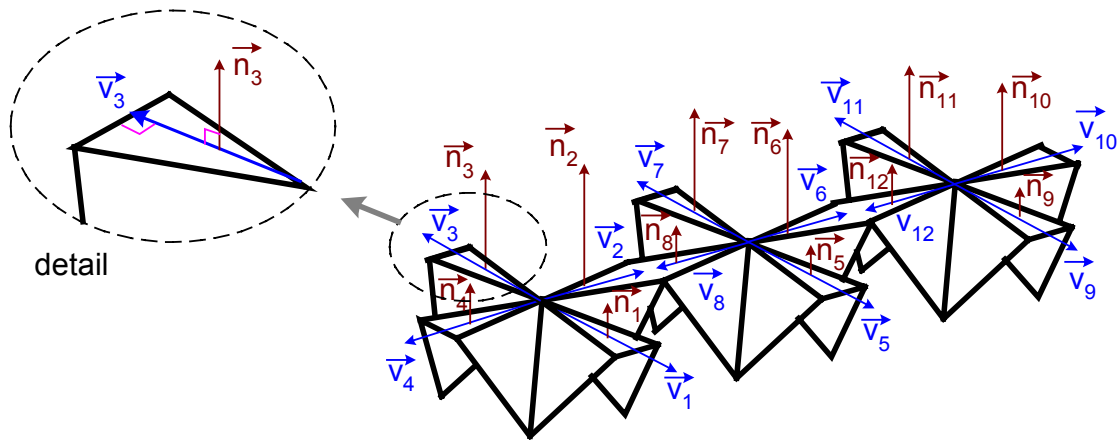


Figure 4.27: The \vec{n} and \vec{v} -vectors

When converting the unit vectors from cartesian to spherical coordinates, the number of unknown variables is reduced. Instead of using the cartesian variables (x,y,z), Figure 4.28 shows how to define a vector using three different parameters: ϕ , θ , and length r . If the vector has unit magnitude, the number of unknowns can be reduced since r always equals one.

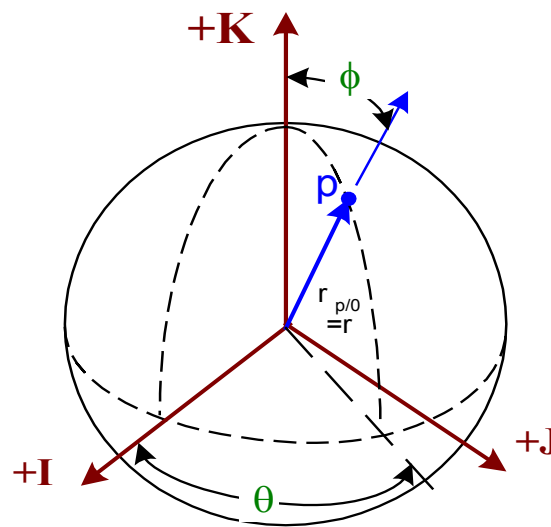


Figure 4.28: Spherical Coordinates

Figure 4.29 shows how the spherical coordinates are used to define the vectors in the detail of Figure 4.27. Note that the linking, triangular face of the unit cell is included in the drawing. The \vec{n}_3 vector is specified by two coordinates, θ_n and ϕ_n similar to the \vec{p} -vector in Figure 4.28. Since the \vec{v} -vectors are always perpendicular to their corresponding \vec{n} -vectors, it is only necessary to specify one parameter for the \vec{v} -vectors. This one parameter appears as the θ_v value in Figure 4.29. θ_v value will change as the matrix twists and deforms. This automatically reduces the number of unknowns the computer needs to find for Method 2. The intermediate coordinate system, $i''j''k''$, will aid in this coordinate transformation. This intermediate coordinate system is defined such that the \vec{n}_3 vector is colinear with the k'' direction and the \vec{v}_3 vector lies in the $i''j''$ plane.

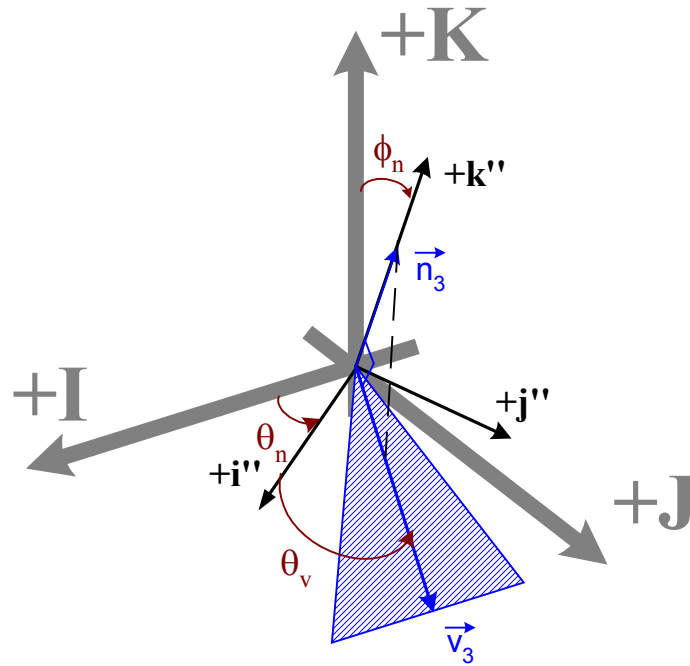


Figure 4.29: Example of Spherical Coordinates

From this explanation for why we using spherical coordinate instead of cartesian coordinates, below are the rest of the principles for Method 2 that uses the spherical coordinates.

4.10 Initial Guess (Method 2: Interpolate and Initial Guess)

From the givens, we can develop the Initial Guess vector of what \vec{n}_i and \vec{v}_i vector values may be by using only spherical coordinates $[\phi, \theta]$. As previously stated, spherical coordinates can reduce the number of variables that needs to be iterated. Originally all \vec{n} unit vectors are $[0, 0, 1]$ in the cartesian coordinate and all \vec{v} unit vectors are in their cartesian directional vector. Equation 4.23 is a sample of the coordinates for the \vec{v} unit vectors of one cell. This sample can be duplicated based upon the numbers of cells in the crust matrix.

$$[\vec{v}_1; \vec{v}_2; \vec{v}_3; \vec{v}_4] = [(0, -1, 0); (1, 0, 0); (0, 1, 0); (-1, 0, 0)] \quad (\text{Equation 4.23})$$

As previously described, one of the givens are the Z_{fij} points, which are the user specified heights of several of the unit cells. The initial guess requires interpolating between any two fixed Z_{fij} points.

Since graphically explaining interpolation for a matrix is too complex, Figure 4.30 is a 2D example of an interpolation for one row.

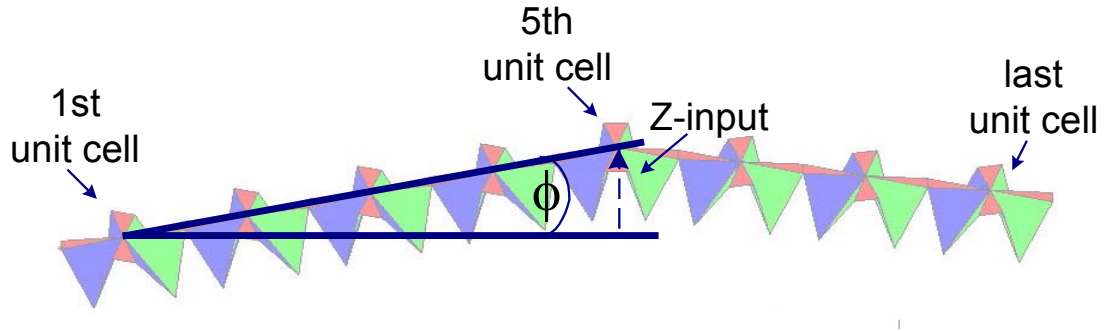


Figure 4.30: Interpolation From Z-input

For Figure 4.30, the first, fifth, and last z-values of the related unit cell are constrained. The fifth z-value is constrained by a Z-height input value. Similar to Method 1, the unit cells in between should follow the straight line between two z-constraints. The difference between the two interpolation styles is in the \vec{n} and \vec{v} vector values. Instead of cartesian coordinates, these values start off as spherical coordinates.

Sub-Step 1:

Calculate the angle of deformation, ϕ , between two Z- constraints.

$$\phi_i = -\sin^{-1} \left(\frac{(Z_f - Z_{f-1})}{(\text{Cell}_f - \text{Cell}_{f-1}) * \text{dist}} \right) \quad (\text{Equation 4.24})$$

$f = 2, 3, \dots \# \text{ of inputs} + 1$

$Z_{f \ i,j} = \text{current fixed Z heights}$

$Z_{f-1 \ i,j} = \text{previous fixed Z heights}$

$\text{Cell}_f \ i,j = \text{The cell where Z heights are currently fixed}$

$\text{Cell}_{f-1 \ i,j} = \text{The cell where Z heights are previously fixed}$

Sub-Step 2:

After calculating the ϕ_i 's all the new \vec{n} 's are

$$\vec{n}_i = \begin{bmatrix} \phi \\ \theta \end{bmatrix} = \begin{bmatrix} \phi_i \\ 0 \end{bmatrix} \quad (\text{Equation 4. 25})$$

The θ are still all zeros because the \vec{n} 's are not rotated laterally only longitudinally. The \vec{v} 's values remain the same before and after deformation. If any of the cells has a fixed x or y cartesian coordinate, the ϕ_i 's for the normal vectors of that cell will not be part of the unknowns in the Initial Guesses vector. For example in Figure

4.31, if the x-coordinate of the center-point for the unit cell shown is fixed, then both the ϕ for \vec{n}_2 and \vec{n}_4 are a fixed value that will not be part of the Initial Guesses vector.

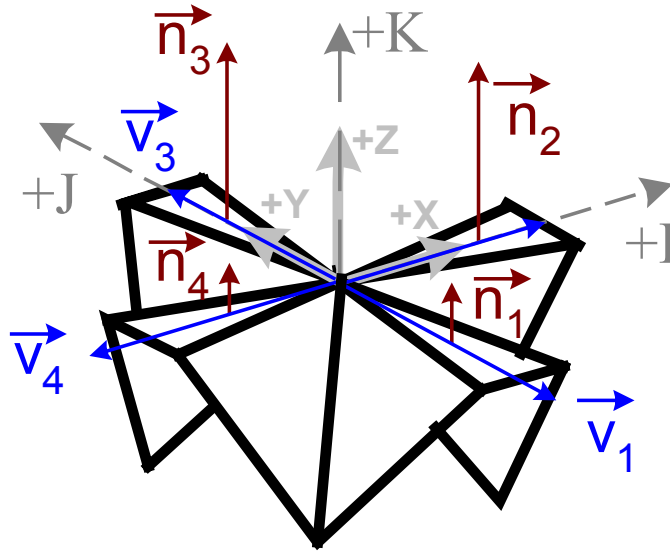


Figure 4.31: \vec{n} and \vec{v} -vectors for One Unit Cell

The unknowns from the Initial Guesses vector will be the variables that the iteration process will be determining for calculating the potential energy in the system with the joint angles.

4.11 Duplications (Method 2: “Create Relationship” Block)

We can take advantage of the symmetries within the matrix of cells to simplify the analysis by deriving the relationship between the cartesian coordinates of the \vec{v} and \vec{n} vectors. These symmetries create duplications of the variables. Below are explanations of the duplications:

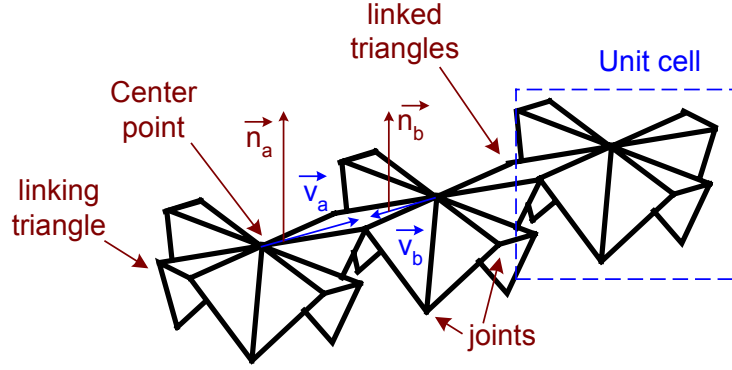


Figure 4.32: Linking unit cells

1) The unit normal vectors of any triangles that are rigidly linked together will always point in the same direction, as seen in Figure 4.32. We can call these values “identical twins.”

a. $\vec{n}_a = \vec{n}_b$ (Equation 4.26)

2) For every \vec{v} -vector that radiates out toward the edge of a linking triangle, there is an equal and opposite \vec{v} -vector on the neighboring cell. We can call these vectors “mirror-image twins”:

a. $\vec{v}_a = -\vec{v}_b$ (Equation 4.27)

Using these properties the number of unknown variables can be reduced: once one variable is found, the twin of the found variable can easily be calculated. When the number of unknown variables is reduced, the number of iterations or guesses that computer has to perform is reduced. This will also increase the calculation speed. The properties can also be used as a check on the accuracy of the guesses. The calculated values can be compared to each other to determine the accuracy of the calculation.

4.12 From Spherical to Cartesian (Method 2: Inside Iteration Box)

Based upon the previous explanation of spherical coordinates and the already explained principles for Method 2, this section will describe how the spherical coordinates unknowns will be converted back to cartesian coordinates to calculate the joint angles. The first step is calculating the cartesian coordinates for \vec{n} vectors.

Finding \vec{n} vectors

As previously stated, the θ_n and ϕ_n parameters for the \vec{n} -vectors are defined in the same way that the spherical coordinates are defined in Figure 4.28. Based upon the generic conversion of spherical coordinates, the equation for the \vec{n} -vectors is as follows (Ginsberg,1998):

$$\vec{n}_i = \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} = \begin{bmatrix} \sin(\phi_n) * \cos(\theta_n) \\ \sin(\phi_n) * \sin(\theta_n) \\ \cos(\phi_n) \end{bmatrix} \quad (\text{Equation 4.28})$$

Finding \vec{v} vectors

An imaginary i'' - j'' - k'' frame was created in Figure 4.29, with the \vec{n} -vector aligned along the k'' -axis and the \vec{v} -vector lying in the i'' - j'' plane. The first task is to find the rotation matrix between the I,J,K frame and the i'' , j'' , k'' frame. This can be accomplished using two rotations as shown in Figure 4.29; first rotating about the J-axis by ϕ_n , then rotating about the K-axis by θ_n . The θ_v is embedded in the \vec{n} -vector rotation as one will see in the following calculation. Figure 4. 33 below will describe the first rotation.

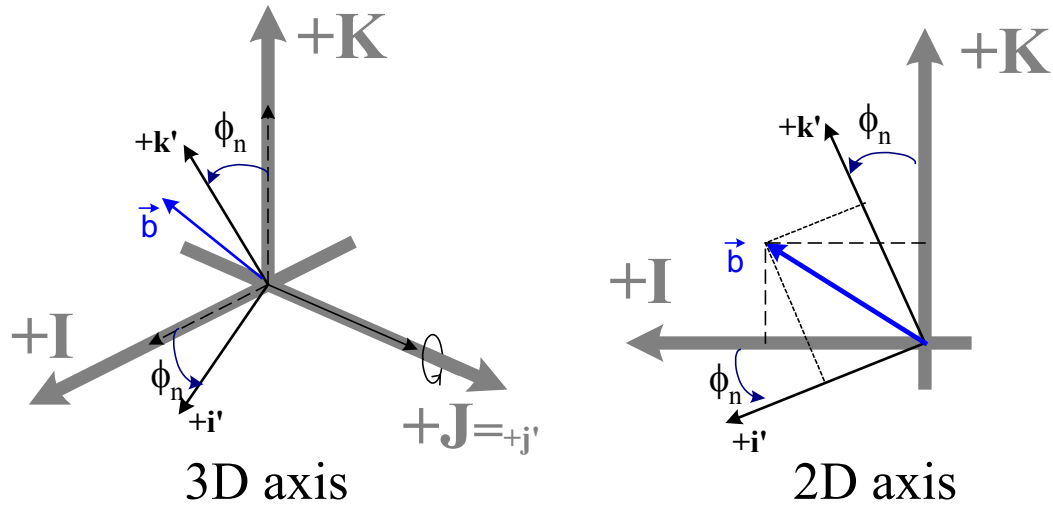


Figure 4.33: Rotation about J-axis

Rotation about the J-axis:

Let $\vec{b} = x\hat{i} + y\hat{j} + z\hat{k}$ denote an arbitrary vector that is being rotated about the J-axis by ϕ_n . After rotating, the new \vec{b} -vector can be referenced back to the space-fixed coordinate system using trigonometry, as shown in the 2D axis image of Figure 4.33.

$$\vec{b} = x * [\cos \phi_n \hat{I} - \sin \phi_n \hat{K}] + y * [1\hat{J}] + z * [\sin \phi_n \hat{I} + \cos \phi_n \hat{K}] \quad (\text{Equation 4.29})$$

The matrix formulation for the rotation of an arbitrary vector about the J-axis is:

$$\begin{bmatrix} \text{space-fixed} \\ \text{coordinates} \end{bmatrix} = \begin{bmatrix} \text{rotation} \\ \text{matrix} \end{bmatrix} * \begin{bmatrix} \text{rotated} \\ \text{coordinates} \end{bmatrix} \Rightarrow \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \cos \phi_n & 0 & \sin \phi_n \\ 0 & 1 & 0 \\ -\sin \phi_n & 0 & \cos \phi_n \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (\text{Equation 4.30})$$

Rotation about the K-axis:

We now derive the rotation matrix for an arbitrary rotation of θ_n about the space fixed K-axis. This rotation is shown in Figure 4.34.

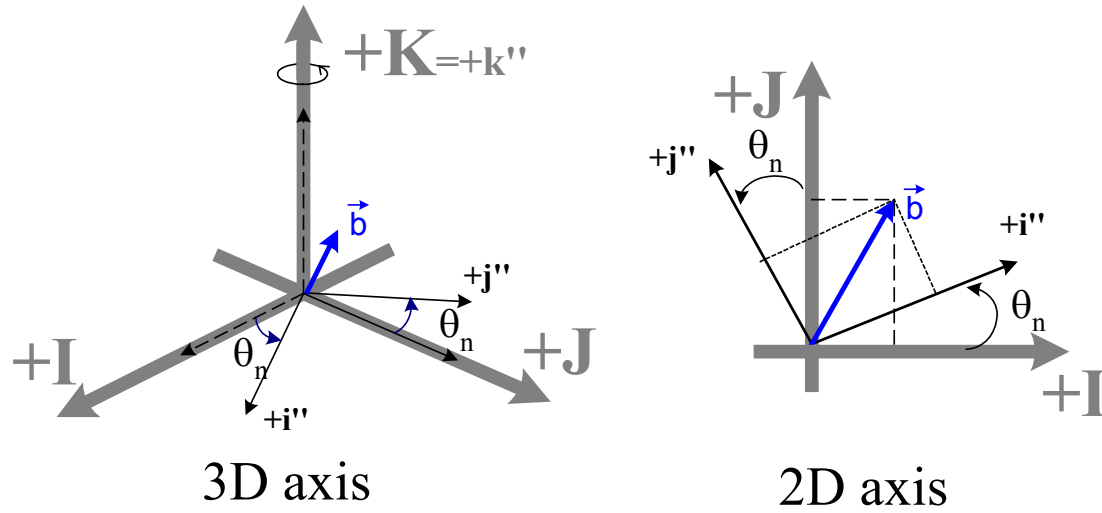


Figure 4.34: Rotation about K-axis

Again using the \vec{b} -vector example: $\vec{b} = x\hat{i} + y\hat{j} + z\hat{k}$, the equation and matrix derivations are as follow:

$$\vec{b} = x * [\cos \theta \hat{I} + \sin \theta \hat{J}] + y * [\sin \theta \hat{I} + \cos \theta \hat{J}] + z * [1\hat{K}] \quad (\text{Equation 4.31})$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (\text{Equation 4.32})$$

After developing the rotation matrices for each axis, the overall rotation matrix can be written as a product of the two rotations. The overall rotation matrix is given as:

$$[R] = [R_z] * [R_y] = \begin{bmatrix} C_{\theta_n} & S_{\theta_n} & 0 \\ -S_{\theta_n} & C_{\theta_n} & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} C_{\phi_n} & 0 & -S_{\phi_n} \\ 0 & 1 & 0 \\ S_{\phi_n} & 0 & C_{\phi_n} \end{bmatrix} = \begin{bmatrix} C_{\theta_n} C_{\phi_n} & -S_{\theta_n} & C_{\phi_n} S_{\phi_n} \\ S_{\theta_n} C_{\phi_n} & C_{\theta_n} & S_{\theta_n} S_{\phi_n} \\ -S_{\phi_n} & 0 & C_{\phi_n} \end{bmatrix}$$

(Equation 4.33)

This is the rotation matrix that converts i'' , j'' , k'' coordinates to I, J, K coordinates.

As a check we can use the \vec{p} -vector, which is parallel to the k'' axis as shown in Figures 4.19 and 4.20. The two rotations that define the \vec{p} -vector and k'' axis are shown in Figure 4.35, and are identical to the two rotations discussed above.

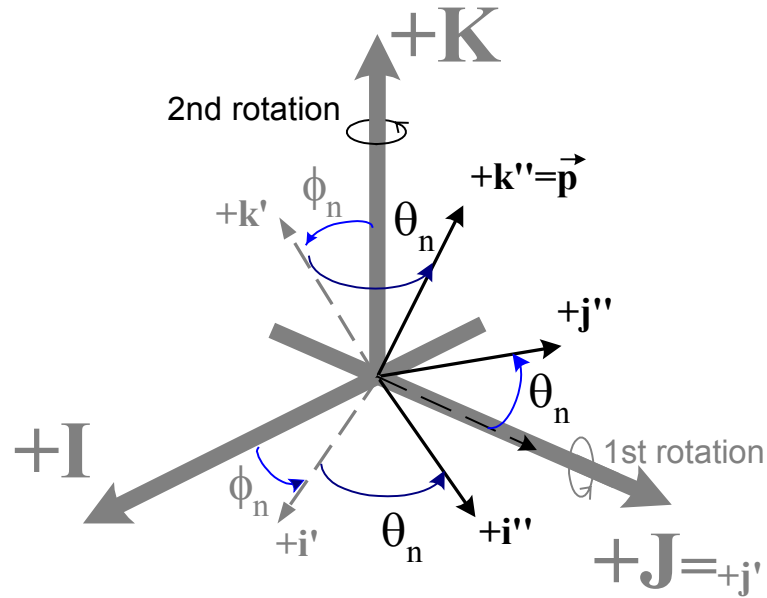


Figure 4.35: Two Rotations

Before being rotated the \vec{p} -vector is parallel to the K-axis. Therefore its rotated value becomes:

$$\begin{bmatrix} C_{\phi_n} * C_{\theta_n} & -S_{\phi_n} & S_{\phi_n} * C_{\theta_n} \\ C_{\phi_n} * S_{\theta_n} & C_{\theta_n} & S_{\phi_n} * S_{\theta_n} \\ -S_{\phi_n} & 0 & C_{\phi_n} \end{bmatrix} * \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} S_{\phi_n} * C_{\theta_n} \\ S_{\phi_n} * S_{\theta_n} \\ C_{\phi_n} \end{bmatrix} \quad (\text{Equation 4.34})$$

The result is the same as in Equation 4.28, for converting from spherical to cartesian coordinates. Also this matrix is checked with vectors other than $[0 \ 0 \ 1]$ and the results is the same as the equation in Ginsberg, 1998. Therefore our rotation matrix is correct.

Based on Figure 4.29, the \vec{v} -vector and the linking triangle lies in the $i''\text{-}j''$ plane. Its position in the i'',j'',k'' reference frame is:

$$[\vec{v}_i\text{'s position}] = \begin{bmatrix} C_{\theta_v} \\ S_{\theta_v} \\ 0 \end{bmatrix} \quad (\text{Equation 4.35})$$

Next, the rotation matrix in equation 4.33 is used to convert the i'',j'',k'' coordinates into I,J,K coordinates.

$$\begin{bmatrix} C_{\phi_n} * C_{\theta_n} & -S_{\phi_n} & S_{\phi_n} * C_{\theta_n} \\ C_{\phi_n} * S_{\theta_n} & C_{\theta_n} & S_{\phi_n} * S_{\theta_n} \\ -S_{\phi_n} & 0 & C_{\phi_n} \end{bmatrix} * \begin{bmatrix} C_{\theta_v} \\ S_{\theta_v} \\ 0 \end{bmatrix} = \begin{bmatrix} C_{\phi_n} * C_{\theta_n} * C_{\theta_v} - S_{\theta_n} * C_{\theta_v} \\ C_{\phi_n} * S_{\theta_n} * C_{\theta_v} + C_{\theta_n} * S_{\theta_v} \\ -S_{\phi_n} * C_{\theta_v} \end{bmatrix} \quad (\text{Equation 4.36})$$

Now we have the equations for converting the \vec{v} -vectors from spherical to cartesian coordinates.

4.13 Calculating the Joint Angles (Method 2: Joint Angles Calculation)

Based upon a given set of cartesian coordinates of the v_i , and n_i vectors, the joint angles can be calculated by applying a series of equations using the geometric design of the unit cells. The joint angle values will be used in calculating the potential energy in the system, which in turn helps calculate the position of the center-points for every unit cell. The mathematical notation is given as:

$$\text{Joint angles: } \delta = G^{-1}(C(v_i, n_i)) \quad (\text{Equation 4.37})$$

$C(v_i, n_i)$ denotes the cartesian coordinates of v_i , n_i , and G^{-1} denotes a set of equations that can take $C(v_i, n_i)$ to find the deformation, δ , of the angles for the unit cells. Below will be a series of equations to analyze an array of unit cells.

Note: This section incorporates some of Paul Bosscher's Master thesis on inverse kinematics. However, this is not the inverse kinematics for this thesis. There is not an inverse kinematics section for this thesis. There is an inverse statics section

4.13.1 Joint Angle Pseudo-code for One Unit Cell

Figure 4.36 shows a unit cell and the detail of the focus area within the dashed circle. The math is as follows:

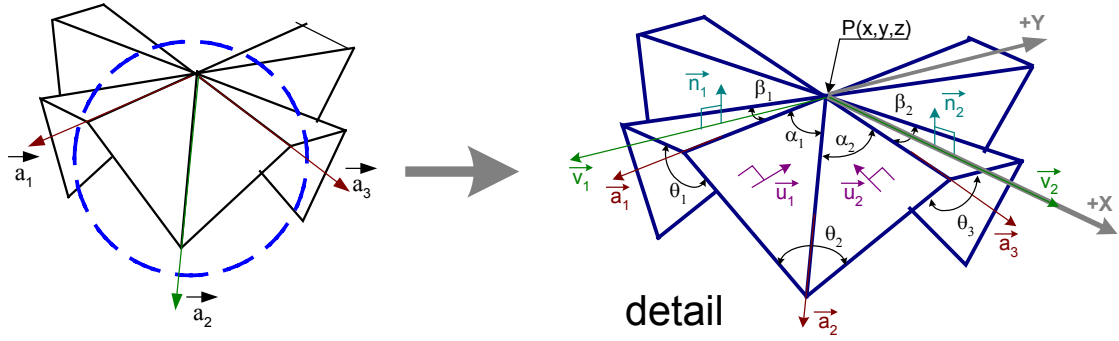


Figure 4.36: Unit Cell

New Definition

- i. \vec{a}_i = Unit vector running along the axis of the revolute joints shown in the Figure.
- ii. \vec{u}_j = Unit vector normal to the plane containing \vec{a}_j and \vec{a}_{j+1}
- iii. β_i = Interior angle of main linking triangle i
- iv. α_k = Interior angle of secondary linking triangle k
- v. θ_j = the angle at joint j

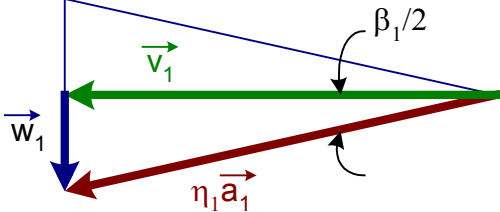
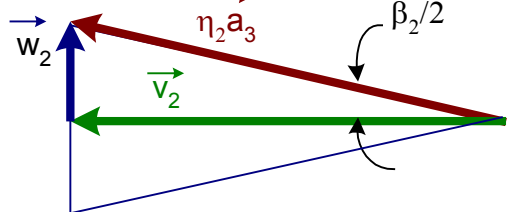
Given: $\vec{v}_1, \vec{v}_2, \vec{n}_1, \vec{n}_2, \alpha_1, \alpha_2, \beta_1, \beta_2$

Find: $\theta_1, \theta_2, \theta_3$

Step 1:

Finding the unit vector \vec{a}_1 and \vec{a}_3 can be achieved by applying Bosscher's equations as seen in Table 4.1 (Bosscher, 2003).

Table 4.1: Finding Unit Vector \vec{a}_1 and \vec{a}_3 from Bosscher's Master's Thesis

 $\eta_1 = \frac{1}{\cos\left(\frac{\beta_1}{2}\right)} = \frac{\ \vec{v}_1\ }{\cos\left(\frac{\beta_1}{2}\right)}$ $\ \vec{w}_1\ = \eta_1 \cdot \sin\left(\frac{\beta_1}{2}\right) \quad (* \text{ see footnote})$ $\eta_1 \cdot \vec{a}_1 = \vec{v}_1 + \vec{w}_1$ $\vec{w}_1 = \eta_1 \cdot \sin\left(\frac{\beta_1}{2}\right) \cdot (\vec{n}_1 \times \vec{v}_1) \quad (*)$ $\vec{a}_1 = \left(\vec{v}_1 + \tan\left(\frac{\beta_1}{2}\right) \cdot (\vec{n}_1 \times \vec{v}_1) \right) \cdot \cos\left(\frac{\beta_1}{2}\right)$	 $\eta_2 = \frac{1}{\cos\left(\frac{\beta_2}{2}\right)} = \frac{\ \vec{v}_2\ }{\cos\left(\frac{\beta_2}{2}\right)}$ $\ \vec{w}_2\ = \eta_2 \cdot \sin\left(\frac{\beta_2}{2}\right) \quad (* \text{ see footnote})$ $\eta_2 \cdot \vec{a}_3 = \vec{v}_2 + \vec{w}_2$ $\vec{w}_2 = -\eta_2 \cdot \sin\left(\frac{\beta_2}{2}\right) \cdot (\vec{n}_2 \times \vec{v}_2) \quad (*)$ $\vec{a}_3 = \left(\vec{v}_2 + \tan\left(\frac{\beta_2}{2}\right) \cdot (\vec{n}_2 \times \vec{v}_2) \right) \cdot \cos\left(\frac{\beta_2}{2}\right)$
---	---

Step 2:

Step 2 describes the math for finding \vec{a}_2 from \vec{a}_1 and \vec{a}_3 by modifying Bosscher's equations. In the set of equations to calculate the joint angles, Bosscher specified using an iterative process by searching through various combinations and checking when

* In the original text, there were errors. This paper is presenting the correct version.

$(\vec{a}_1 \times \vec{a}_3) \cdot \vec{a}_2 \leq 0$. This process does work for finding \vec{a}_2 but it involves solving a nonlinear system of equations.

Another approach is using a direct method by considering the unit cell as a combination of geometric shapes. This method is faster, simpler without having conditions and constraints, and more accurate.

In reference to Table 4.1, Figure 4.37 shows another technique for finding the \vec{a}_2 vector within the dash circle of the previous figure.

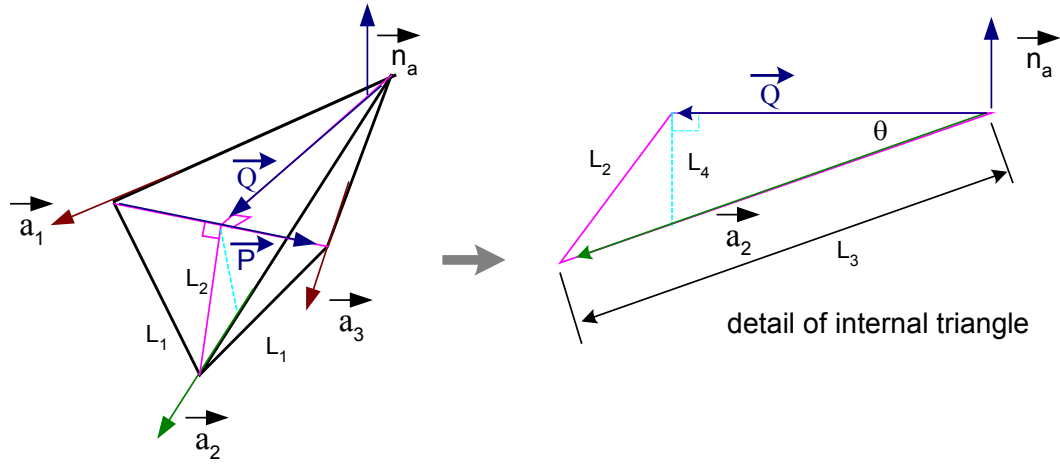


Figure 4.37: Detail of Dashed Circle of unit cell

- 1) Given from Geometry: L_1 , L_3 , L_a (the length of the edge collinear with \vec{a}_1)
- 2) Find \vec{a}_2

Sub-step 1:

Finding the \vec{Q} vector from \vec{P} vector using vector summation rule

$$\vec{P} = L_a \cdot \vec{a}_3 - L_a \cdot \vec{a}_1 \quad (\text{Equation 4.38})$$

$$L_a \cdot \vec{a}_1 + \frac{\vec{P}}{2} = \vec{Q} \quad (\text{Equation 4.39})$$

Sub-step 2:

Finding θ by combining the Pythagorean Theorem and the Law of cosine

$$L_2 = \sqrt{L_1^2 - \left\| \frac{\vec{P}}{2} \right\|^2} \quad (\text{Equation 4.40})$$

$$L_2^2 = \left\| \vec{Q} \right\|^2 + L_3^2 - 2 \cdot \left\| \vec{Q} \right\| \cdot L_3 \cdot \cos \theta \quad (\text{Equation 4.41})$$

$$\Rightarrow \theta = \cos^{-1} \left[\frac{-(L_2^2) + \left\| \vec{Q} \right\|^2 + L_3^2}{2 \cdot \left\| \vec{Q} \right\| \cdot L_3} \right]$$

Sub-step 3:

Find \hat{n}_a which is the unit normal vector to a plane shared by both \vec{a}_1 and \vec{a}_3

$$\vec{n}_a = \vec{a}_1 \times \vec{a}_3 \quad (\text{Equation 4.42})$$

$$\hat{n}_a = \frac{\vec{n}_a}{\left\| \vec{n}_a \right\|} \quad (\text{Equation 4.43})$$

Sub-step 4:

Calculate the last unknown length using trigonometry

$$L_4 = \left\| \vec{Q} \right\| \cdot \tan \theta \quad (\text{Equation 4.44})$$

Sub-step 5:

After finding all the other unknowns calculate the \vec{a}_2 by the vector summation rule and applying the unit vector equation. Figure 4.38 is another detail image from Figure 4.37 that will provide a visual explanation for the following calculation.

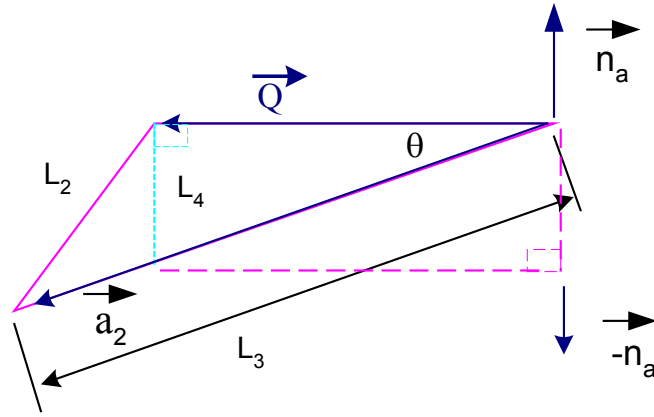


Figure 4.38: Second Detail

$$\vec{a}_2 = -L_4 \cdot \hat{n}_a + \vec{Q} \quad (\text{Equation 4.45})$$

$$\vec{a}_2 = \frac{\hat{a}_2}{\|\vec{a}_2\|} : \text{the unit vector that points in the direction of the middle vector.}$$

$$(\text{Equation 4.46})$$

Step 3:

After finding \vec{a}_2 using this method, the joint angles, θ 's can be found by applying the rest of Bosscher's method, as presented in Table 4.2.

Table 4.2: Finding the Joint Angles θ 's from Bosscher's Master Thesis

<p>Finding \vec{u}_1 and \vec{u}_2:</p> $\vec{u}_1 = \frac{(\vec{a}_1 \times \vec{a}_2)}{\ \vec{a}_1 \times \vec{a}_2\ }$ $\vec{u}_2 = \frac{(\vec{a}_2 \times \vec{a}_3)}{\ \vec{a}_2 \times \vec{a}_3\ }$	<p>Finding θ_1, θ_2, and θ_3:</p> $\vec{u}_1 \cdot \vec{n}_1 = \cos(\pi - \theta_1)$ $\vec{u}_2 \cdot \vec{n}_2 = \cos(\pi - \theta_3)$ $\vec{u}_1 \cdot \vec{u}_2 = \cos(\pi - \theta_2)$ $\theta_1 = -\cos^{-1}(\vec{u}_1 \cdot \vec{n}_1) + \pi^1$ <p>where if</p> $(\vec{u}_1 \times \vec{n}_1) \cdot \vec{a}_1 = 0 \quad \text{then} \quad \theta_1 = \pi$ $\theta_3 = -\cos^{-1}(\vec{u}_2 \cdot \vec{n}_2) + \pi$ <p>where if</p> $(\vec{u}_2 \times \vec{n}_2) \cdot \vec{a}_3 = 0 \quad \text{then} \quad \theta_3 = \pi$ $\theta_2 = -\cos^{-1}(\vec{u}_1 \cdot \vec{n}_2) + \pi$
---	--

¹ In the original equation, finding θ requires the design variable, λ . For this case, the design variable is not necessary.

4.13.2 Joint Angle Calculation for an Array

For an array of unit cells the math is similar to the one unit cell case. The goal is to find the θ 's of every joint for every unit cell in the matrix. The derivation is identical to the one shown in the previous section. The only difference is that now the equations must be compiled into vectors to store the information for the multiple cells in the matrix. Figure 4.39 is a composite of previous images and explanation to remind the reader of what this section is about.

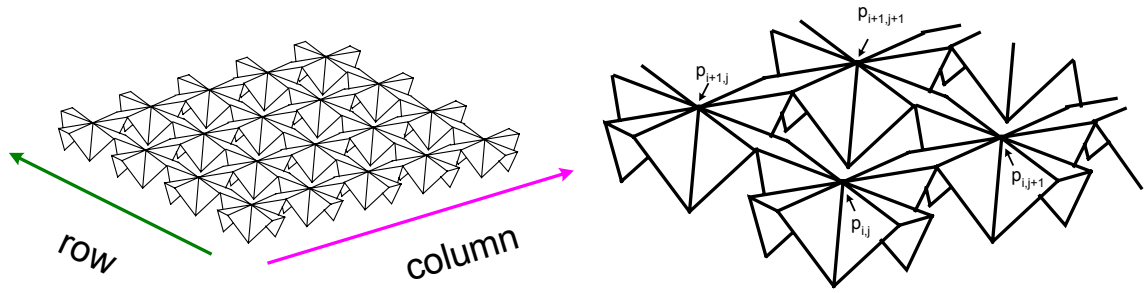


Figure 4.39: Array of Unit Cells with Center points

1) New Definition:

- i) L_a = length of the edge coincident with the \vec{a}_1 -vector from Figure 4.37
- ii) L_{amid} = length of the edge coincident with the \vec{a}_2 -vector from Figure 4.37

$\alpha_1, \alpha_2, \beta_1, \beta_2$ = geometric constants as seen in Figure 4.36

2) Given: $\vec{v}_i, \vec{n}_i, \alpha_1, \alpha_2, \beta_1, \beta_2, k_1, k_2, \text{dist}, L_a, L_{amid}, \alpha_1, \alpha_2, \beta_1, \beta_2$

3) Find: $\theta_{1p}, \theta_{2p}, \theta_{3p}$

- i) Where $p = 1, 2, \dots, 4*n*m$, identical to the definition in section 4.1.2.

Based upon the geometry and Bosscher equations for one cell, we will assume the following:

$$\alpha_1 = \alpha_2 = \alpha \quad (\text{Equation 4.47})$$

$$\beta_1 = \beta_2 = \beta \quad (\text{Equation 4.48})$$

The equations for deriving the \vec{a} -vectors are shown in Table 4.1 and will not be repeated here. However, for computational purposes we will change the previously developed notation for \vec{a} -vectors. The \vec{a} -vectors between the linking triangles will be denoted as $\vec{a}_{\text{mid } i}$, while the \vec{a} -vectors on the edges of the linking triangles are denoted as \vec{a}_i as seen for one unit cell in Figure 4.40.

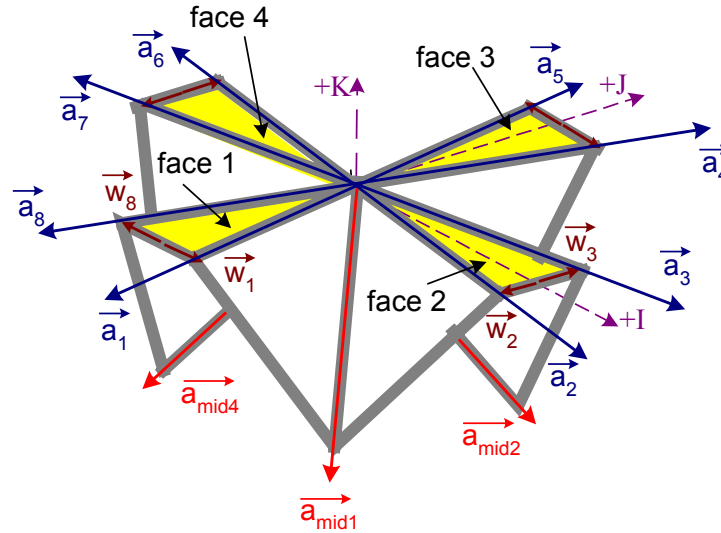


Figure 4.40: Arrangement of \vec{a} and \vec{a}_{mid} vectors

Note this departs from our previous notation.

The \vec{a}_i -vectors are arranged in a matrix as seen below:

$$\vec{a} = \begin{bmatrix} \vec{a}_{1\text{cell}1} \\ \vec{a}_{2\text{cell}1} \\ \vec{a}_{3\text{cell}1} \\ \vdots \\ \vec{a}_{8\text{cell}1} \\ \vec{a}_{1\text{cell}2} \\ \vec{a}_{2\text{cell}2} \\ \vec{a}_{3\text{cell}2} \\ \vdots \\ \vec{a}_{8\text{cell}m*n} \end{bmatrix} \quad (\text{Equation 4.49})$$

After finding all the \vec{a} -vectors, use Table 4.2 to calculate the θ_1 , θ_2 , and θ_3 for each unit cell. The only difference is there will be a subscript variable “p” to indicate which joint it refers to. Since the θ_1 , θ_2 , and θ_3 appear 4 times in each cell the “p” index will range from 1 through $4*m*n$.

4.14 Potential Energy (Method 2: Inside Iteration Box)

From the angles calculated, we can apply the potential energy equation that was already mentioned in Equation 4.22 under the pseudo-code of Method 2. It will not be repeated here. The coordinate constraints also add potential energy values to the total sum of the Potential Energy. If the iteration process produces coordinate results that deviates from the constraint coordinates, the iteration process will be “penalized” with a very high potential energy value as seen in Equation 4.22. The penalty k_3 value in Equation 4.22 is

insignificant as long as this value is at least 2X greater than any of the other 2 k values. For this reason, we arbitrarily selected 100,000 as the k_3 value. If the potential energy is high, the iteration process to re-evaluate the guesses until the energy value is at its lowest state. Also consider there is an additional stiffness value on any fixed Z coordinate. For example in Figure 4.30, the fifth cell has a Z-height input. Therefore the potential energy equation will have an additional term with the difference between the value in which the fifth Z should be fixed at Z_{fij} , and the value in which the computer calculated, Z_{fij}' . This additional stiffness will reduce the likelihood of the iteration process from deviating from the given value as seen in Equation 4.22.

4.15 Ending Comments for Method 2

After finding the spherical coordinates for the \vec{n} and \vec{v} vectors, the vectors can be converted back to cartesian coordinates using one of the previously mentioned principles. From the cartesian coordinates of the \vec{n} and \vec{v} vectors, the position can be calculated by applying inverse statics. Below will be the explanation of forward and inverse statics that provide the overall structure for Method 2. The flow chart that will later be mentioned in the statics section can be used in parallel with the already mentioned flow chart to clarify Method 2.

4.16 Forward and Inverse Statics: Overall Statics of Method 2

In Method 2, both the inverse and forward statics are applied to complete the statics circle and to solve for the position coordinate values of the center-points of the unit cells. Figure 4.41 is a diagram that describes the relationship between the two types of statics.

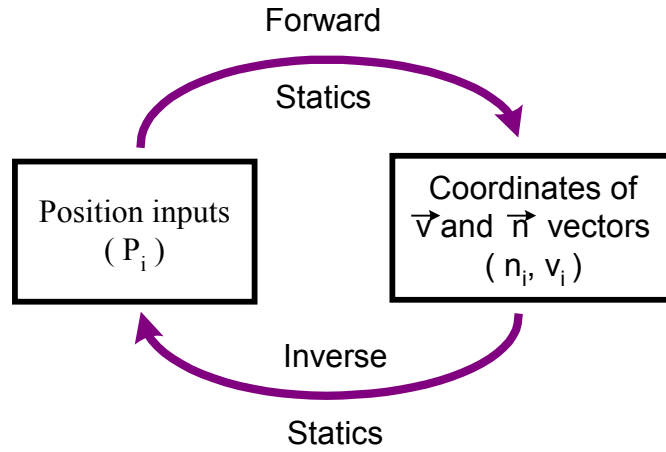


Figure 4.41: Forward and Inverse Statics

First the user will input a set of cartesian coordinates for the center-points of some unit cells. Because of the manipulation of several z-values and the rigid linking between any two unit cells, the coordinate position values will not be accurate. From the inputs, Method 2 will apply forward statics to determine the coordinates of the \vec{n} and \vec{v} -vectors. From this calculation, Method 2 will also apply inverse statics to correctly recalculate the positions. This is an overall description of the relationship between the two.

For a more detail version of how Method 2 calculates the positions by applying an iterative process, refer to Figure: 4.42.

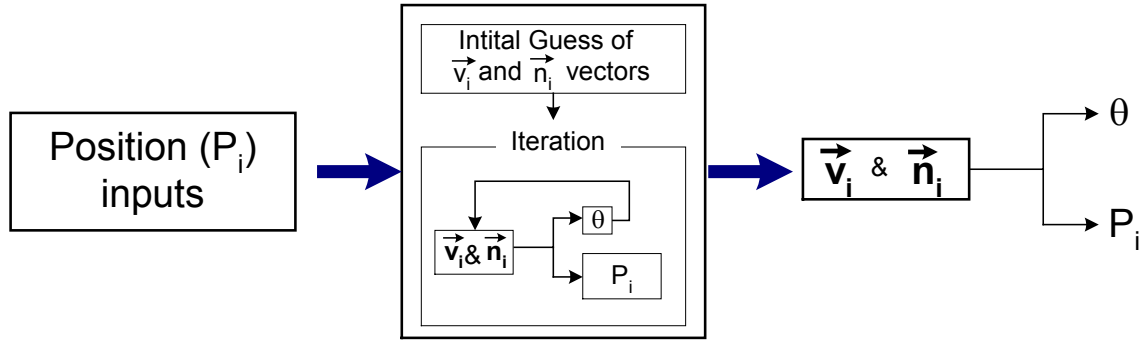


Figure 4.42: Calculates the Position

The user will input a set of cartesian coordinate position values for the center-points for some of the unit cells in the matrix. Embedded inside the methods are additional coordinate constraints. As stated above, the cartesian coordinate values are not accurate. There is a direct relationship between the positions of the unit cells and the \vec{n} and \vec{v} -vectors. At this point, we also do not know the correct coordinate values for \vec{n} and \vec{v} -vectors. However, we can create initial guesses of what those values can possibly be. As stated above, the \vec{n} and \vec{v} -vectors can be calculated by using an iterative process. The iterative process will calculate \vec{n} and \vec{v} -vectors by attempting to minimize the potential energy of the angles from the joints within each unit cell. After producing the cartesian coordinates of \vec{n} and \vec{v} -vectors, a correct list of the values for the position of the center-points can be calculated along with the angles in the joints. The cycle is complete.

In the following sections, the mathematics for both statics will be explained.

4.16.1 Inverse Statics

The inverse statics will be described first because forward statics applies inverse statics to iteratively arrive at the \vec{n} and \vec{v} -vectors values. The mathematic notation is shown:

$$C(P_i) = G(n_i, v_i) \quad (\text{Equation 4.50})$$

We need a series of equations for the \vec{n}_i and \vec{v}_i unit vectors denoted by $G(n_i, v_i)$. From this we can calculate the cartesian coordinate positions of the center-points denoted by $C(P_i)$. The inverse statics can also be applied to develop the inputs for calculating the angles, which in turn helps forward statics to determine the cartesian coordinate positions of the \vec{n}_i and \vec{v}_i unit vectors. However, the main purpose of inverse statics is to calculate the position values of the center-points of every unit cell in the matrix.

4.16.1.1 Inverse Statics Equations

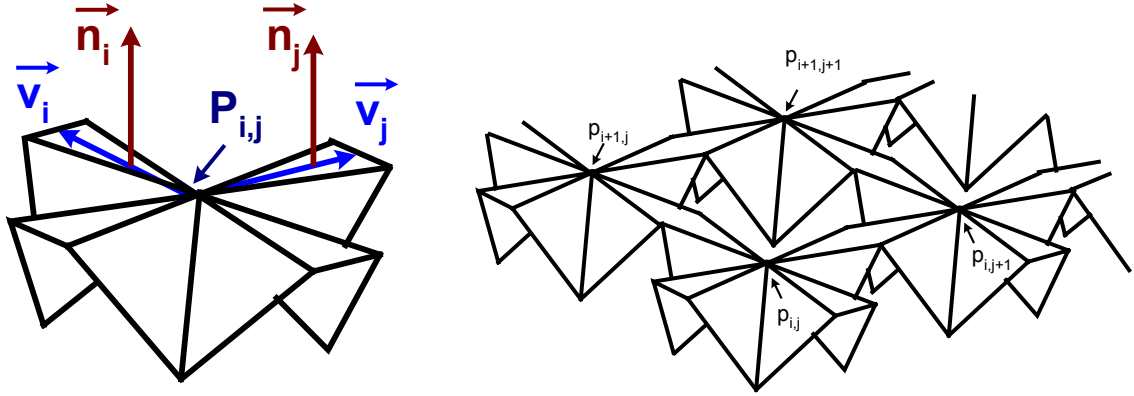


Figure 4.43: Inverse Statics Diagram

From Figure 4.43, the image on the left describes the \vec{n} and \vec{v} -vectors for one of the cells in a matrix. By following the direction of the \vec{v} unit vectors a certain distance, one can calculate the cartesian coordinate position values of any of the center-points. For example, we know that at the first cell, the position value is $(0,0,Z_{F1,1})$. We can follow the \vec{v}_i unit vector a “dist” length. The arrival point is the next centerpoint at $P_{i+1,j}$ as shown on the right image. We can continue onto the next \vec{v} unit vector until we arrive at the next center-point. Of course we can also go the other direction and arrive at $P_{i,j+1}$. This process can continue until we know all of the center-points.

The equations for the center-points are as follow:

$$P_{i,j} = \begin{bmatrix} \text{Row}_i \\ \text{column}_j \end{bmatrix} = \begin{bmatrix} \vec{V}_{i,j} * \text{dist} + P_{i-1,j} \\ \vec{V}_{i,j} * \text{dist} + P_{i,j-1} \end{bmatrix} \quad (\text{Equation 4.51})$$

After knowing the center-points position, the cartesian coordinates of every vertex of every unit cell can be calculated by following the $\vec{a}_{1,2,3}$ unit vectors for the $P_{i,j}$ cell as shown in Figure 4.44. The vertices are referring to the end of the edges of which the \vec{a}_1 , \vec{a}_2 , and \vec{a}_3 vectors follow.

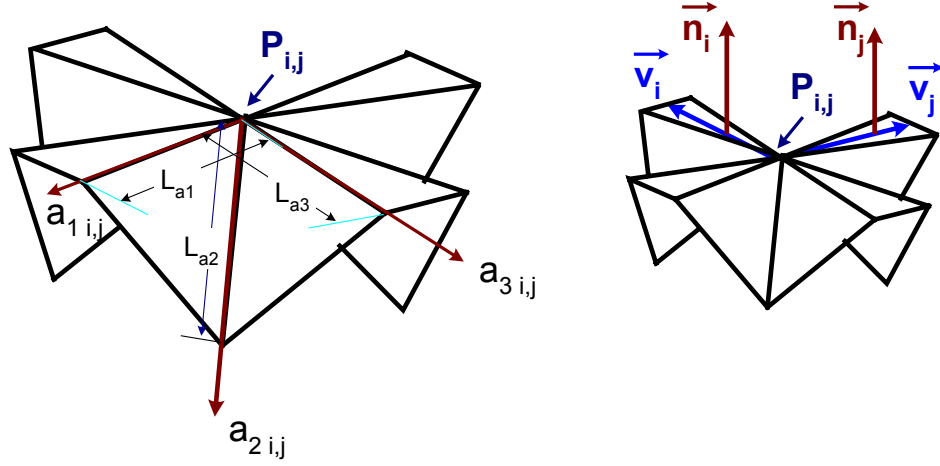


Figure 4.44: Calculating the Edge Vertices

The equations for the vertices are as follow.

$$\vec{a}_{i,j} = \begin{bmatrix} \vec{a}_{1(i,j)} * L_{a1} \\ \vec{a}_{2(i,j)} * L_{a2} \\ \vec{a}_{3(i,j)} * L_{a3} \\ \vdots \\ \vec{a}_{11(i,j)} * L_{a11} \\ \vec{a}_{12(i,j)} * L_{a12} \end{bmatrix} + P_{i,j} \quad (\text{Equation 4.52})$$

$\vec{a}_{i,j}$ = A group of $\vec{a}_{i,j}$ vertices. For each unit cells there are twelve unit $\vec{a}_{i,j}$ -vectors.

L_{ai} = length for the corresponding $\vec{a}_{i,j}$ vectors.

4.16.2 Forward Statics

Based upon all the inputs of the cartesian coordinate position values of all the unit cells, all the \vec{n} and \vec{v} -vectors can be calculated. The mathematical notation for the forward statics is given as:

$$C(n_i, v_i) = G(P_i) \quad (\text{Equation 4.53})$$

The $G(P_i)$ denotes the sets of equations that take on the given position values input, P_i . $C(n_i, v_i)$ is the resulting cartesian coordinates of the \vec{n}_i and \vec{v}_i unit vectors. The implementation of forward statics is applied in the Method 2: Under-constrained Actual Manufacturable Crust Model. Since the forward statics for the crust matrix problem cannot be solved using a series of equations, we are going to apply an iterative process. The iterative process uses the inverse statics to calculate the angles based upon the initial guesses of the \vec{n}_i and \vec{v}_i unit vectors for every linking triangle on every unit cell. Next it will calculate the potential energy from the angles. Then it will take the energy calculated from the angles and modifies the \vec{n}_i and \vec{v}_i unit vectors. Forward statics is accomplished. The iterative process is being accomplished in “Fmincon”: a pre-packaged minimization function in MATLAB. Below is the math for the set-up.

4.16.2.1 Forward Statics Algorithm

Figure 4.45 is three unit cells rigidly linked together with the center-points labeled as $P_{i,j}$. The \vec{n}_i and \vec{v}_i unit vectors are labeled based upon the referencing convention mentioned earlier.

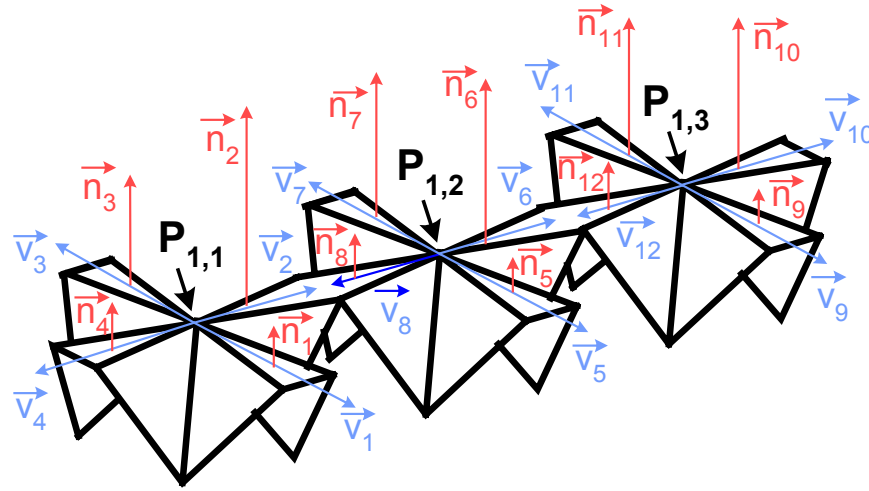


Figure 4.45: Linking Unit Cells

There are 4 \vec{n}_i and 4 \vec{v}_i unit vectors for every $P_{i,j}$ value. For forward statics, we want to derive \vec{n}_i and \vec{v}_i vector values from the $P_{i,j}$ values. Since they are unit vectors, at this point it does not matter where they are located in space. As previously mentioned, there is not a direct method to calculate \vec{n}_i and \vec{v}_i vector values. We will be applying an iterative process to solve the problem. The pseudo-code is very similar to Method 2. Most of the steps will be referenced back to Method 2.

Given: Same as Method 2

Find: \vec{n}_i and \vec{v}_i unit vectors

Minimize: Potential Energy (Equation 4.22 from Method 2)

Step 1:

Determine the initial guess of the \vec{n} and \vec{v} -vectors.

Step 2:

Apply the spherical coordinates rotation matrix to find the cartesian coordinates of the \vec{n} and \vec{v} -vectors.

Step 3:

Apply the inverse statics equations to calculate the \vec{n} and \vec{v} -vectors.

Step 4:

Apply the Joint Angle calculation to determine the joint angles.

Step 5:

Calculate the potential energy in the system with the additional constraints on the unit cell.

Step 6-8:

Repeat steps 2-5 by changing the resulted spherical coordinate guesses back into cartesian coordinates, placing back into the iterative process applied by pre-package

program 'Fmincon' and modifying guesses. Continue iterating to search for the optimal combination of spherical coordinates for the lowest potential energy.

Step 9:

Convert the final spherical coordinates back to cartesian coordinates of the \vec{n} and \vec{v} -vectors. From here we have finished the forward statics.

Step 10:

Next will be the inverse statics for taking cartesian coordinates of the \vec{n} and \vec{v} -vectors and finding the position values of the center-points for each unit cell.

4.17 Unknowns, Equations, and Degrees-of-Freedom

The number of unknowns differs for methods 1 and 2. We will use several examples to show how we will count up the numbers of unknowns.

The number of equations is a count of the independent equations that each method uses to solve the problem. However, this does not count any of the equations that are used to derive these relevant equations. The number of equations differs for each example of Method 1 as one will later see, but stays the same for all examples of Method 2.

For any rigid body, there are 6 Degrees-of-Freedom (DoF): 3 translational and 3 rotational. As rigid bodies are attached together to create one deformable body, most of the rigid bodies lose their translational DoF. We can count each translational DoF by the direction in which the rigid body can move in the X, Y, and Z direction. The convention we will be using to analyze the rotational DoF is the “Roll-Pitch-Yaw” rotations for the rods and unit cells. “Roll-Pitch-Yaw” is a transformation convention for a rigid body rotating about the Z, Y, and X axes respectively. To visualize this convention, Figure 4.46 shows the hull of a boat in water and how it can move about the axes.

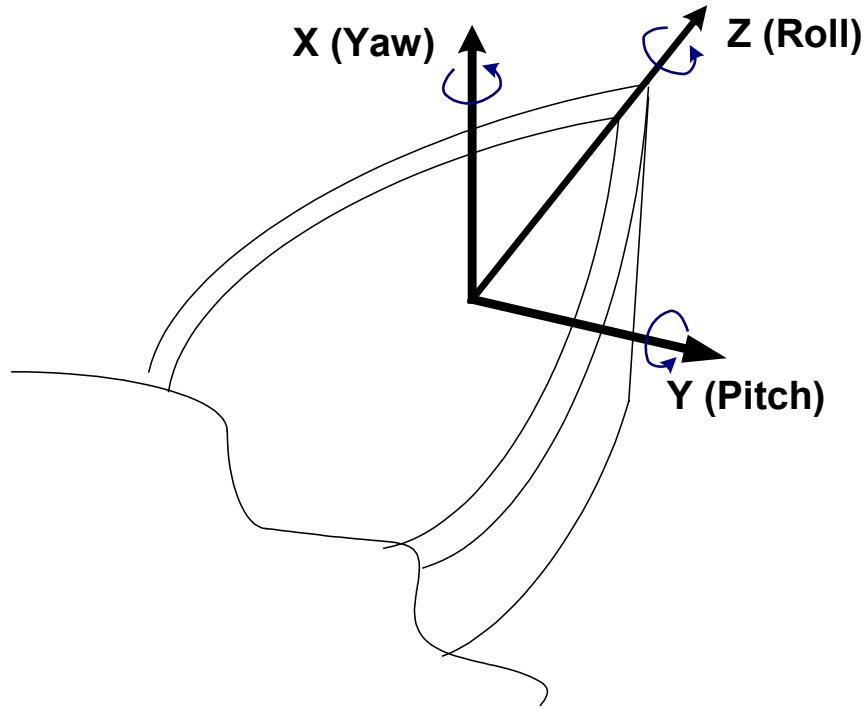


Figure 4.46: Roll-Pitch-Yaw

For this DoF analysis, we will not remove any DoF to fix the crust matrix in space. It is up to the controls department to determine how to attach the crust matrix to the Digital Clay base. This in turn will affect the number of DoF that will be removed. For both methods, we have enough length constraints and coordinate constraints to fix the matrix in space for analyzing and calculating.

4.17.1 Method 1

Below are the analyses for Method 1 using a simple line example and a matrix example.

Number of Unknowns for a Serial Chain

For the first method, the number of unknowns is the number of cartesian coordinates in the matrix because this is what we are solving for. For a 1-by-7 matrix of unit cells as seen in Figure 4.47 the number of unknowns is (3 Cartesian Coordinates)*(# unit cells)=**21 unknowns**.

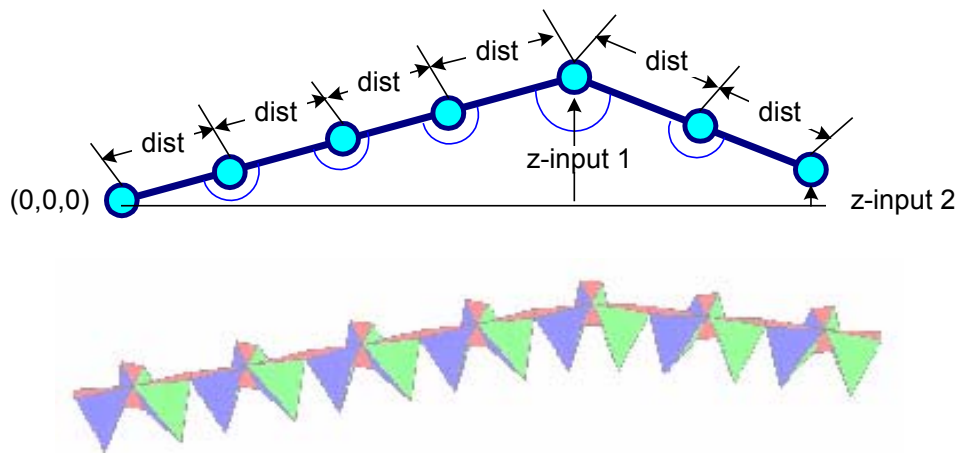


Figure 4.47: Numbers of Unknowns and Constraints for a 1-by-7 Matrix

Number of Equations for a Serial Chain

The additional fixed coordinate values, such as the Z-heights, and the constant-length equation are constraints that limit the searching process. For every constraint, there

is an equation. The constraints do not affect the number of unknowns for Method 1. There is an additional equation, the potential energy minimization equation. For Figure 4.47, the constraints are the first coordinate points at (0,0,0), the Z-height inputs, any other constrained coordinate values, and the length “dist” constraints. Therefore for Figure 4.47 with an example of a 1-by-7 matrix, there are 5 coordinate constraints + 6 length “dist” constraints + 1 minimization equation = **12 equations total**.

Since there are constraints that affects the unknowns, this is not minimum number of variables that determine the exact position of the unit cells. In other words, the number of unknowns does not equal the degrees-of-freedom.

Degrees-of-Freedom for a Serial Chain

First we will start with one rod having 5 DoF as seen in the first image in Figure 4.48 between point A and B. That means that rod A-B has 3 (translational) + 2 (rotational) = 5 DoF. Note that in the figure, the balls are the endpoints for the rods. There are only two rotational DoF for rod A-B because if rod A-B rotates about its own axis, it is still a cylinder. This means from the “Roll, Pitch, and Yaw” definition of rotation, we have ignored the “Roll”. In the middle image, another rod is added. Again this rod starts off with 5 DoF. Since rod B-C is attached to a fully defined rigid body, rod B-C loses 3 translational DoF. Therefore rod B-C has 2 DoF left. We can continue adding rods to the chain as seen in the last image. By the previous argument, each added rod only contains 2 DoF for a linearly connected chain. Again we are not going to fix this

matrix. It is beyond the scope of this thesis to determine how the crust matrix will be fixed to the Digital Clay device. **Therefore we will keep the matrix floating in space.**

For a 1-by-7, the DoF is: $[5+2+2+2+2+2](\text{total DoF}) = 15 \text{ DoF for a 1-by-7.}$

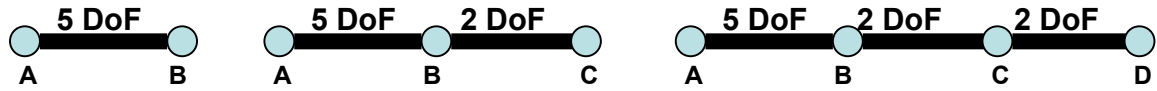


Figure 4.48: Method 1 2D Example for DoF

Number of Unknowns for a Matrix

For a matrix, the number of unknowns gets more complex. As the size of the matrix increases, the unknowns increase. For the case of a 4-by-5 as seen in Figure 4.49, there are a total of 20 unit cells. That means there are $(3 \text{ Cartesian Coordinates}) * (20 \text{ Unit cells}) = 60 \text{ unknowns.}$

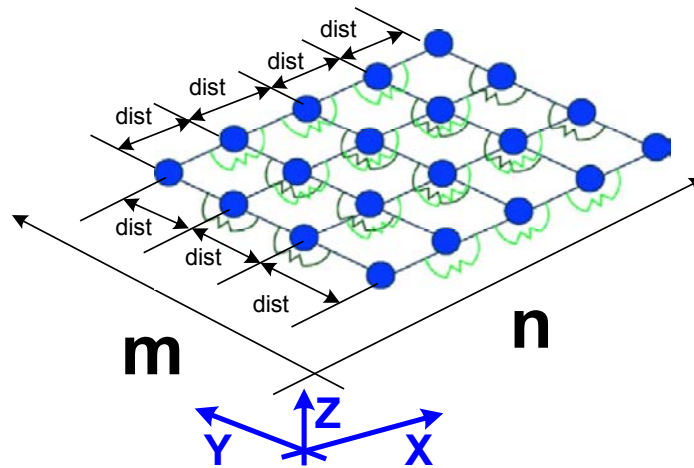


Figure 4.49: 4-by-5 Matrix of Constraints

Number of Equations for a Matrix

The number of coordinate constraints equations is dependent on the number of user inputs to constrain the X, Y, or Z coordinate values of the unit cell centers. In Figure 4.49, we know there are $[(n-1)*m + (m-1)*n]$ (length constraint equations) + 1 minimization equation = 32 equations. The coordinate constraints are not counted because this value is based upon the user inputs and varies for different situation.

Degrees-of-Freedom for a Matrix

To determine the DoF for a matrix, first let's look at a simple case with a 2-by-2 matrix. Again the first rod has 5 DoF as seen in Figure 4.50. The second rod is attached to Rod A-B and adds 2 DoF, similar to the case shown in Figure 4.48. Rod C-D initially has two DoF, similar to Rod A-C. But Rod D-B and Rod C-D have to meet at one connecting point D. The locus of points equidistant from the two points B and C is a circle. Therefore the DoF for both Rod C-D and D-B is one. In other words, the location of point D is the last element that needs to be known to define this 2-by-2 matrix. Once this is known, we can total up the DoF: $(5+2+1) = 8$ DoF.

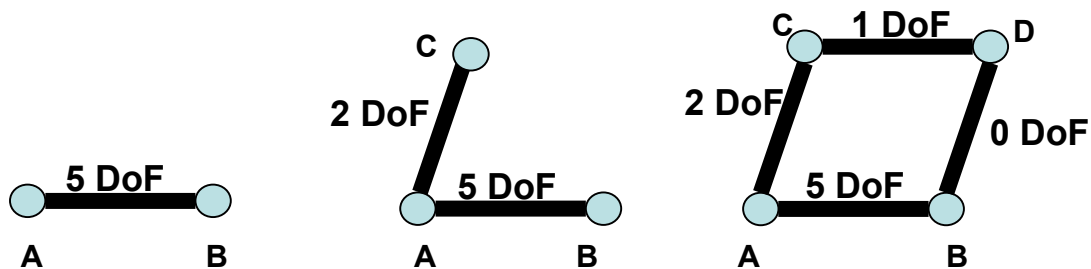


Figure 4.50: 2-by-2 DoF Example

For the case of a 4-by-5 as seen in Figure 4.49, this same addition is applied. The Degrees-of-Freedom: $5 + 2*(n-2) + 2*(m-1) + 1*(m-1)*(n-1) = 29$ DoF for a 4-by-5 matrix using the Method 1 modeling technique.

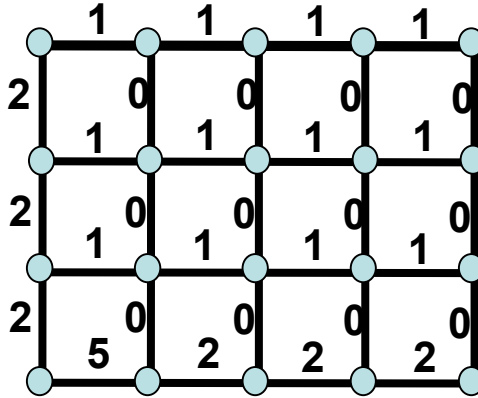


Figure 4.51: Method 1: 4-by-5 DoF Example

Note that the previous equation holds for any matrix of m rows and n columns.

4.17.2 Method 2

Below is the analysis for Method 2 with the unit cells.

Number of Unknowns for a Serial Chain

For Method 2, the unknowns are the \vec{n} and \vec{v} -vectors. It is the cartesian coordinates of these vectors that determines the centerpoints and joint angles for minimizing the potential energy. As previously stated, we were able to reduce the number of unknowns by using spherical coordinates for the \vec{n} and \vec{v} -vectors. If any one of the faces of any unit cell is fixed, then \vec{n} and \vec{v} -vectors for that particular face are known.

Face fixing will be further explained in the next chapter, but not accounted for in this section. However, the other constraints such as the coordinate constraints and the length constraints will not affect the numbers of unknowns. This is because they are imbedded in the energy equation. For any unit cell, there are 4 θ_n , 4 ϕ_n , and 4 θ_v unknown values. If any of these variables is on the same plane or on a linked triangle, then the number of unknowns is further reduced because there are duplications in the θ_n , ϕ_n , and θ_v . The realization of duplication was explained earlier in section 4.11.

For example, for a 1-by-7 matrix of unit cells as seen in Figure 4.47, there are (m rows)*(n columns)* (4 θ_n + 4 ϕ_n + 4 θ_v per unit cell) = 84 unknowns. If an X number of the faces is fixed, then we can subtract X θ_n , X ϕ_n , and X θ_v unknown values from the 84 unknowns. Lets say one of the faces in the matrix is fixed, then that leaves 84 – [1 θ_n , 1 ϕ_n , and 1 θ_v unknown values] = 81 unknown values left. Of course for this analysis we assume that no face is being fixed. That means the number of unknowns is still 84. Now we will apply the realization of duplication method because some of the θ_n , ϕ_n , and θ_v share the same linking triangles. There are 12 linking triangles that are rigidly connected for a 1-by-7 matrix. These linking triangles are described by 12 θ_n , 12 ϕ_n , and 12 θ_v . However, half of these are duplicated. That means there are only 6 θ_n , 6 ϕ_n , and 6 θ_v unknown values. So finally we have 84- 6(linked triangles)*3 (the variables that are being duplicated) = **66 unknowns**.

Number of Equations for a Serial Chain

The number of equations for Method 2 is similar to Method 1= [number of coordinate constraints] + [number of length constraints] + 1 [energy minimization equation]. For the 1-by-7 matrix in Figure 4.47, there are 4 coordinate constraints + 6 length “dist” constraints + 1 minimization equation = **11 equations total**. This number is the same as Method 1 because the constraints are the same for both methods. As previously explained the coordinate constraints equation and the minimization equation can be combined into one equation. The length constraints can be considered after the iteration process. Of course this is all implementation of the equations and does not affect the total number of equations for Method 2.

Degrees-of-Freedom for a Serial Chain

Unlike Method 1, all rigid bodies now start off with 6 DoF. If they are linked together end-to-end then each additional body will lose 3 translational DoF. What is left for the connecting rigid body is the “Roll-Pitch-Yaw” rotation. For Method 2, the shape of the rigid body is a triangle. Notice that in contrast to Method 1, we will count the “Roll” as part of the rotational DoF, because a triangle rotated about the Z-axis changes its orientation.

Figure 4.52 shows how the counting convention starts and is propagated for a linear case. The first triangle still has all of its 6 DoF.

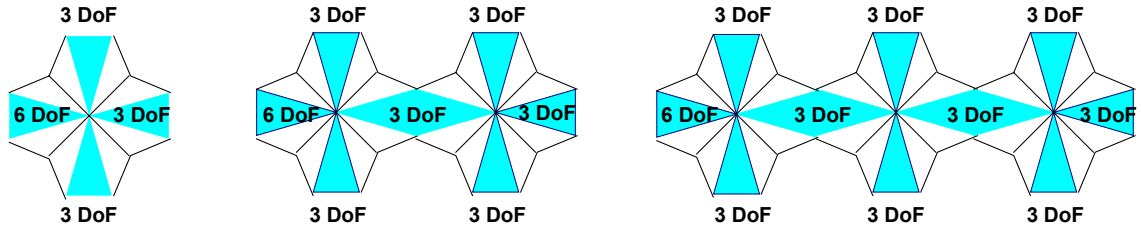


Figure 4.52: Method 2: DoF Counting

The other three triangles in the first image lose their translational DoF when they are connected to the first triangle. They are left with 3 rotational DoF. The middle image shows two triangles linking, that means these two triangles share 3 DoF because these triangles become 1 rigid body when they are rigidly connected to each other. The last image shows how the DoF propagates. For a **1-by-7 matrix, there are actually: $6+3*3(\text{DoF per face})*7(\text{unit cells}) = 69 \text{ Degrees-of-Freedom}$.**

If one compares the number of unknowns and the Degrees-of-Freedom, one will realize the difference is 3. In the previous example for a 1-by-7, there are 66 unknowns and 69 DoF. The reason for the difference is because in the DoF analysis there are 6 DoF for the first face while there are only 3 unknowns for the first face. There is a positive difference of 3: 6 DoF and 3 unknowns. Therefore, we can safely say that there is a direct relationship between the DoF and the number of unknowns. For a matrix of any size we will not need to count up the number of unknowns. We just count the DoF and then subtract 3. **We will later derive the number of unknowns for a m-by-n matrix from the Degrees-of-Freedom.**

Number of Equations for a Matrix

For a 4-by-5 matrix, there are $[(n-1)*m + (m-1)*n]$ (length constraint equations) + 1 minimization equation = 32 equations. Similar to Method 1, the coordinate constraints are not counted.

Degrees of Freedom for a Matrix

Calculating the Degrees-of-Freedom for Method 2 is similar to how we calculated for Method 1, except we do not remove the “Roll”. Figure 4.53 shows one triangle fully defined by 6 DoF. The other connected triangles in the cell have 3 rotational DoF because they have lost their 3 translational DoF. The second image is a propagation of the first. When Cell C is connected to Cell A in the third image, there are only 3 DoF between the two. However when Cell D is connected, there are only 2 DoF for C-D. The reason for 2 DoF is similar to the previous argument for Method 1 shown in Figure 4.48. However, now the linked triangle can also “Roll” between point C and D. The last linked triangle, between D and B, has only 1 DoF because it can only “Roll”.

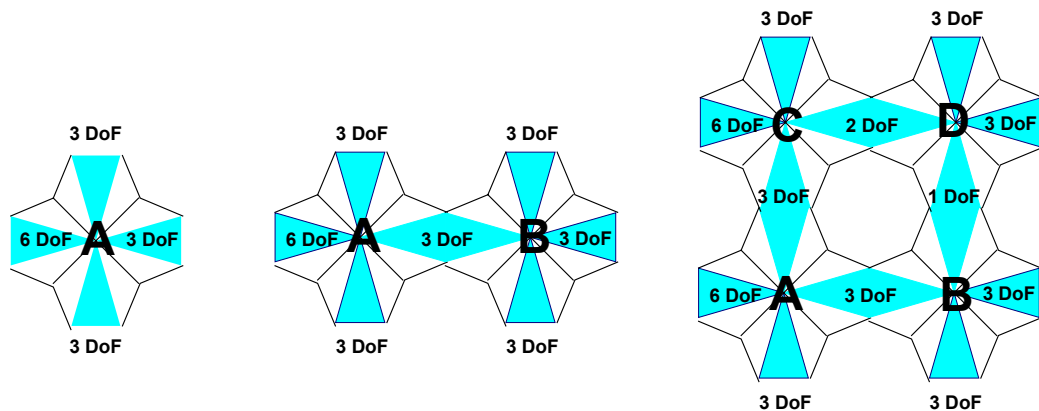


Figure 4.53: Method 2: 2-by-2 DoF Example

For a 4-by-5 matrix, the counting technique for the Degrees-of-Freedom can be propagated for the whole matrix as shown in Figure 4.53.

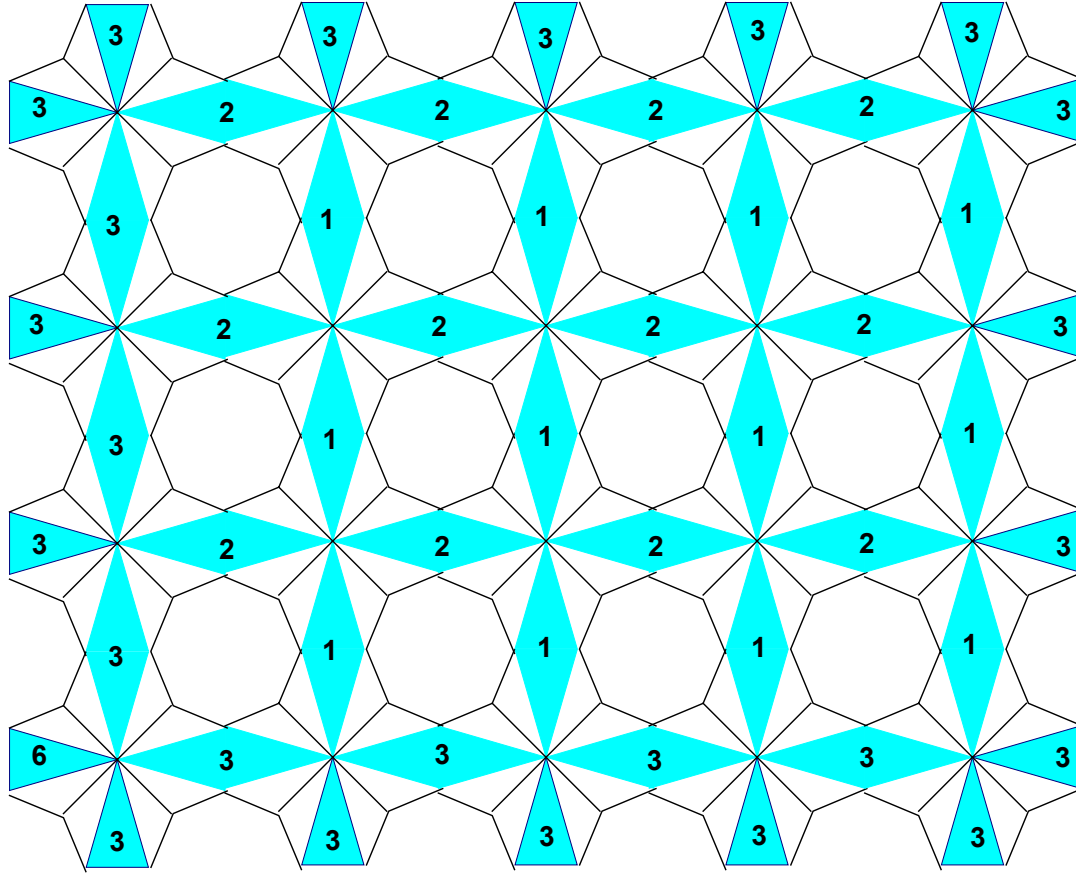


Figure 4.53: Method 2: 4-by-5 DoF Example

The DoF counting technique for Method 2 is very similar to Method 1. If we compare Figure 4.53 to Figure 4.51, the main difference is that there is one more Degree-of-Freedom for every rigid body in Method 2 as compared to Method 1. The reason is that we did not remove the “Roll” for Method 2.

The equation for the DoF is:

$$DoF = 6 + F * Te * n + Se * m - F + 3 * (n - 1) + 3 * (m - 1) + 2 * (m - 1) * (n - 1) + 1 * (m - 1) * (n - 1) \quad (\text{Equation 4.54})$$

F=3 DoF for every triangle faces

Te= 2 Top edges

Se= 2 Side edges

From this equation, the **DoF for a 4-by-5 matrix = 30+24-3+12+9+24+12= 114 DoF .**

Number of unknowns for a matrix

Previously we explain in the section about “*Degrees-of-Freedom for a Serial Chain*”, we can calculate the numbers of unknowns from the DoF by subtracting 3. **The number of unknowns is 111.**

4.17.3 Ending Remarks for Section

By comparing the statics of the two methods, we have a far better understanding of the two different methods. The first method does not attempt to reduce the unknowns by calculating the Degrees-of –Freedom first. However, it is still faster, because overall there are fewer unknowns and less Degrees-of-Freedom than the second method. The

second method considers every aspect of the actual deformable crust matrix—there should be more unknowns and DoF.

Note: For counting the number of DoF for the whole deformable body, we have considered both external and internal DoF. External refers to the DoF the structure would have if it behaved like a rigid body and did not deform. Internal refers to the DoF that cause the body to deform, and does not include any external DoF. For an example for the difference between external and internal DoF, consider a box with a hinged lid. The whole box in space has 6 external DoF because it can rotate and translate without any hindrance to its motion. The hinge on the box only has 1 internal DoF because it can only flip up and down. Therefore the box has 6 external and 1 internal DoF. The total DoF for this hinged box is 7.

The number of equations for Method 1 and 2 are the same because the constraints applied are the same.

4.18 Ending Remarks for Chapter

The two methods mentioned are developed for calculating the position of the center-points of the every unit cell position. Below are various comments about the methods.

4.18.1 Benefits

How would these two methods benefit anyone? These two methods mentioned in the previous sections will benefit the Digital Clay team members because the methods can be applied for both shape display and shape editing-- the original goals of the Digital Clay project.

As previously explained, shape display is when the Digital Clay crust matrix can be computer-commanded to acquire a wide variety of desired shapes. In both methods that were discussed in the earlier sections, the user can input several constraints and the programs that implemented the methods would calculate and display the shape based upon the applied constraints. This will benefit the Digital Clay team members because they can predict the shape of the crust matrix based upon the inputs. This also allows the members to determine what material will be best fitted for manufacturing the crust based upon the inputs and the shape display.

Shape editing is when the user can modify an existing shape into another shape. A former example of shape editing is the car hood model idea introduced at the beginning

of this chapter. The car hood Lotus design can morph into a Ferrari then into a Corvette as seen in Figure 4.5: Morphing of the Car Hoods. To accomplish this shape-editing task, the final position of the Lotus will become the initial stage of the Ferrari. And so on. This morphing of shapes has already been accomplished when the flat surface (an initial stage) of the crust matrix has morphed into the Lotus (final stage). This morphing process can be further enhanced for different shape morphing when there is a loop command in the programs. This loop command has not been implemented yet due to the time constraint for this thesis.

Another bonus to the two methods is that these two methods can be piggybacked to increase speed and accuracy. Since the first method is faster, the user inputs will be placed in the first method. The final results of the first method can then be placed into the second method for improving the accuracy of the shape displayed. This piggybacking idea will be further explained in the next chapter.

There are other benefits from this project to the sub-groups of the Digital Clay team. These will also be mentioned in the last chapter.

CHAPTER 5

EXPERIMENTS AND RESULTS

Below are the results from both methods, implemented by MATLAB programming. The first method is successfully executed to produce the deformation for a line or an m -by- n matrix. The speed of convergence dramatically increases when the initial guesses are improved with linear interpolation. The fixed points are the user specified center-points.

The second method implementation currently produces a line. It may take up to two more months to complete the coding for the implementation of the second method to produce a complete matrix. Presently, another master student in the Computer Science department is considering completing the coding in C++. This collaboration will be further discussed in the future works section of the last chapter. For the second method, there are actually two programming versions. The difference between the two versions is in the constraints. The versions will be described in details in the next section. In total there are three programs: one for the first method, and two for the second method.

In this chapter, the results from the first method will be compared to both of the versions from the second method. From these results, Method 1 outperforms Method 2 in the time comparison test and is competitive in the accuracy of output values. The Method 1 will be used to create several matrices to determine the time and accuracy of the implementation for a matrix. Finally, the different car-hood models will be presented.

The time it takes to produce the final results is determined on the computer being used. Below is a table describing the computer/s that will be running the programs.

Table 5.1: Computer Configuration

Dell Computers from Mechanical Engineering CAE Clusters		
<i>Processor</i>	<i>Display Adapter</i>	<i>Network Adapters</i>
Intel (R) Xeon TM CPU 3.20 GHz	NVIDIA Quadro FX 500	3Com EtherLink XL 10/100 PCI Intel(R) Pro/1000 MTW Network Connection

As a reminder, the **units will not be shown** because it is assumed that all the values of the results have the same units or in the same family of units. The values of the results are not as important as the relations of the values to each other.

5.1 Compare and Contrast

As previously stated there are actually three different programs. The first program is for the first method, which is an abstract model of the crust matrix with one stiffness value. Therefore the resulting graphs are only stick figures with the vertices representing the center-points of each unit cell. It is not necessary to show every unit cell, because we are mainly interested in the center-points locations. The second and third programs are different versions of the second method. Method 2 consists of two different stiffness values for the two different joint designs for each unit cell. There will be 2 graphs from these programs: first will show the unit cells deforming and the other will show the center-points. The reason for the unit cell graphs is because the angles between any two faces will deform as the position of the center-points changes. The first method does not deal with the angles within each unit cell.

The first version of Method 2 has one of the faces fixed horizontally- in this case the face at the edge of the first unit cell for a line. Refer to Figure 5.1 for a visual description.

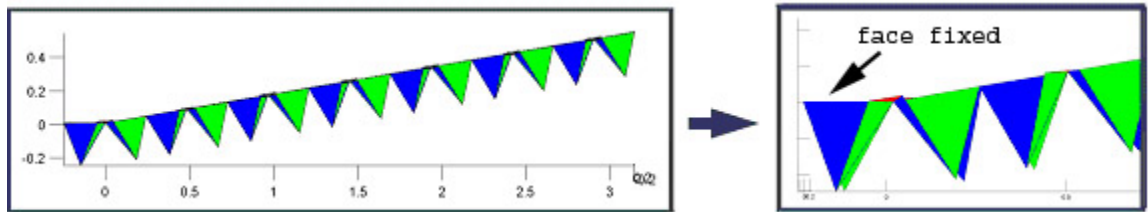


Figure 5.1: Fixed Face: Line of Cells (Left) and Detail of Fixed Face (Right)

There is an assumption that the crust may be fixed at the edges to the future walls of the digital clay interactive device.

The second version of the second method has several center-points fixed at the user specified heights. Therefore the faces are not fixed to any assumed edges as seen in Figure 5.2 with the two images.

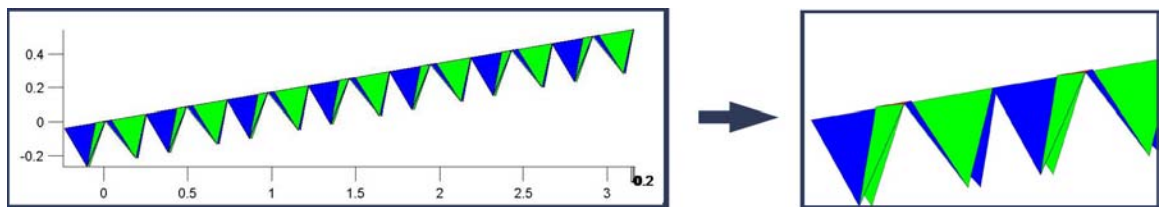


Figure 5.2: Free Face: Line of Cells (Left) and Detail of Free Face (Right)

From the three programs that implemented the different methods, the results are compared based upon the center-points, the computational time, the number of iterations, and the energies that are calculated from the configuration of the lines. Also joint angles and the energy in the joints angles are compared. Since there are a lot of joint angles for any one case, only the third line test will show the joint angle test. The third line test is the most complicated in comparison to the first and second test. The sections below are divided up based upon the test. We are currently studying a line because it is visually easier to compare the difference among the programs. The line will be at an arbitrary size of 1-by-7—not too small to see the difference and not too large so that we have to wait for hours for the results. In later sections we will show the matrix of center-points.

5.2 Line Test 1

Below are the results for the first line test—a line with 1 input at the end of the line at 0.5 unit.

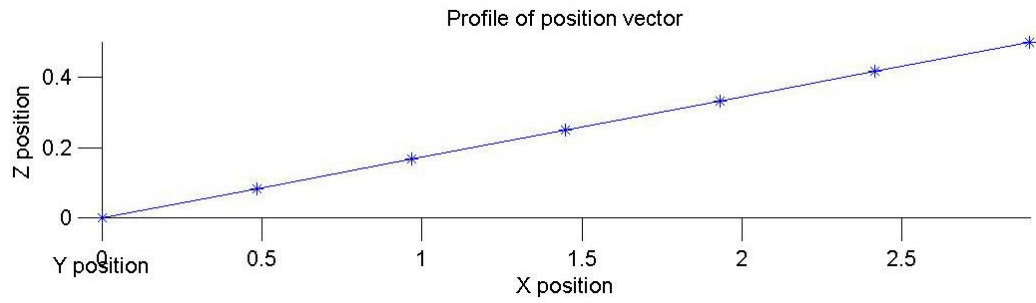


Figure 5.3: Line Test 1 Method 1: Abstract Model

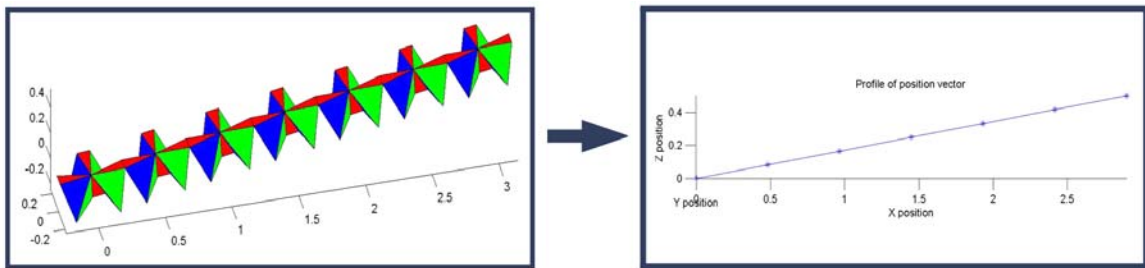


Figure 5.4: Line Test 1 Method 2: With Fixed Face

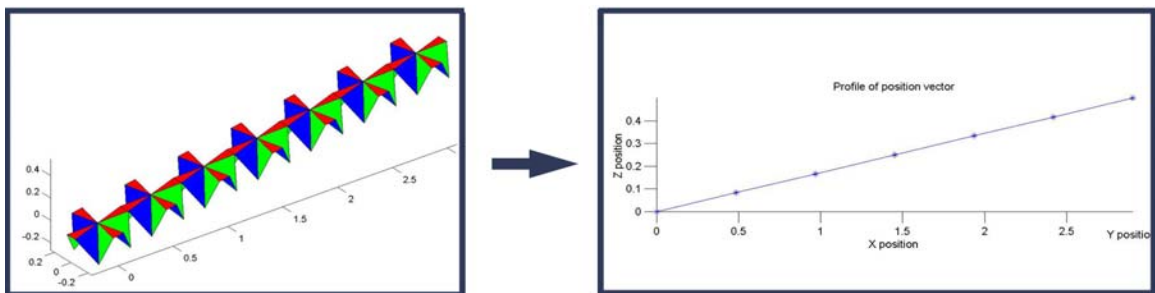


Figure 5.5: Line Test 1 Method 2: Without Fixed face

For this example, the numbers of unknowns, numbers of equations, and the Degrees-of-Freedom are shown in the Table 5.1.

Table 5.2: Kinematics Comparison Test 1

	# Unknowns	# Equations	Degrees-of-freedom
Method 1	21	11	15
Method 2: Fix Face	63	11+3=14	66
Method 2:Free Face	66	11	69

From Table 5.1, for Method 1 there are $3(\text{unknown coordinates value}) * 7(\text{unity cells}) = 21$ unknowns. There are $4(\text{coordinate constraints}) + 6(\text{length constraints}) + 1(\text{minimization equation}) = 11$ equations total. There are 15 DoF. The explanation for DoF was already explained in the previous chapter and will not be repeated here.

For Method 2: Fix Face, there are $[3(\text{spherical } \vec{n} \text{ and } \vec{v}\text{-vectors values}) * 4(\text{faces for each cell}) * 7(\text{unit cells in the matrix})] - [6(\text{duplicates}) * 3(\text{spherical } \vec{n} \text{ and } \vec{v}\text{-vectors values})] - 3(\text{spherical } \vec{n} \text{ and } \vec{v}\text{-vectors values that are fixed on one of the faces}) = 63$ unknowns. There are also $63+3=66$ Degrees-of-Freedom. Again, we will not repeat why we added 3 to the number of unknowns to find the DoF. The number of equations is the same as Method 1 except there are now 3 more equations because one of the face is fixed.

For Method 2: Free Face, there are $[3(\text{spherical } \vec{n} \text{ and } \vec{v}\text{-vectors values}) * 4(\text{faces for each cell}) * 7(\text{unit cells in the matrix})] - [6(\text{duplicates}) * 3(\text{spherical } \vec{n} \text{ and } \vec{v}\text{-vectors values})] = 66$ unknowns. Three more unknowns than the Fix Face example, because no face is fixed. There are also $66+3=69$ Degrees-of-Freedom. Again there are 11 equations, the same number of constraints as Method 1.

Just from the figures, one can tell that the results for all three cases are very similar. However, let's look at a closer detail by actually comparing the differences. Below are the tables for comparing the various results. As a reminder, Method 1 is the abstract model. Method 2 is the actual manufacturable model with the first face being fixed. Another version of Method 2 is without any face being fixed.

Table 5.2 shows the time it takes for the any methods to actually converge to the answers.

Table 5.3: Time Comparison for Line Test 1

	Time
Method 1	0.404 sec
Method 2: Fix Face	202sec ~ 3.4 min
Method 2:Free Face	5.9 sec

Obviously the first method is faster mainly because it requires less unknown variables to iterate and also there is only one stiffness value to consider. Following first is the Method 2 without the Fix Face and then in last place Method 2 with Fix Face.

The next table compares the number of iterations that it takes for the methods to converge and the stored energy value at convergence. Again, the units are not the significant factor since they all have the same units. Only the values are shown.

Table 5.4: Iteration and Energy Comparison for Line Test 1

	Iteration	Energy
Method 1	8	3.270×10^{-9}
Method 2: Fix Face	68	70.104
Method 2:Free Face	74	59.302
Method 1- Method 2: Fix Face	60	70.104
Method 1- Method 2: Fix Face	66	59.302

From the table, it takes fewer iterations amount for Method 1 to converge. Also there is less energy stored in Method 1, because there are fewer angles to consider for summing up the potential energy in the system. Coming in second is Method 2: Free Face with last place Method 2 with the fixed face.

The third table compares the Z-values. Along with the table is a graph of the results from the three programs.

Table 5.5: Z-values Comparison

Z-values Results	
M1 – M2:fix	M1 – M2:free
0	0
0	0
0.001	0
1.000×10^{-4}	0
0	0
1.000×10^{-4}	0
1.000×10^{-4}	0

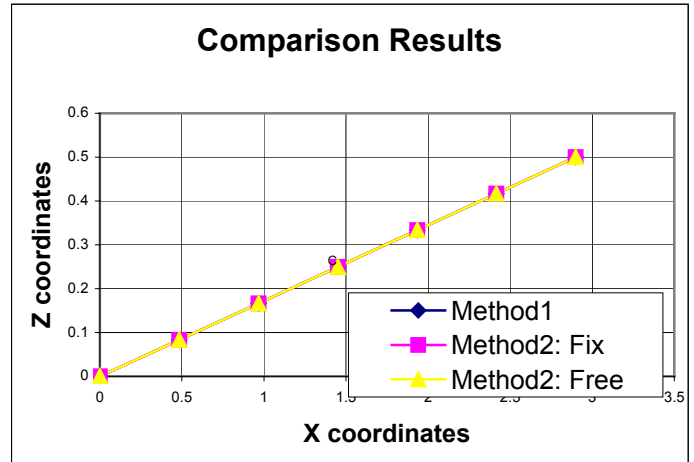


Figure 5.6: Z-Values Results for Line Test 1

From the table and the graph, all three programs turn out to have the same results. The differences in the Z-values from the results of Method 1 and 2 are so small that they are negligible. It appears there is not a difference between the Method 1 and 2: Free Face. Even in the graph, all the results appear to overlap, giving the graph the appearance of containing only one line.

Ironically so, it takes longer for both versions of Method 2 to converge to an answer but their results are very similar to Method 1, which takes less than half a second. Of course this is only a simple case with only 1 input.

5.3 Line Test 2

Now we will add two inputs: 0.5 unit at the beginning and 0.5 unit at the end. This is a simple test, but the results should show us how each program can handle the first center-point being displaced.

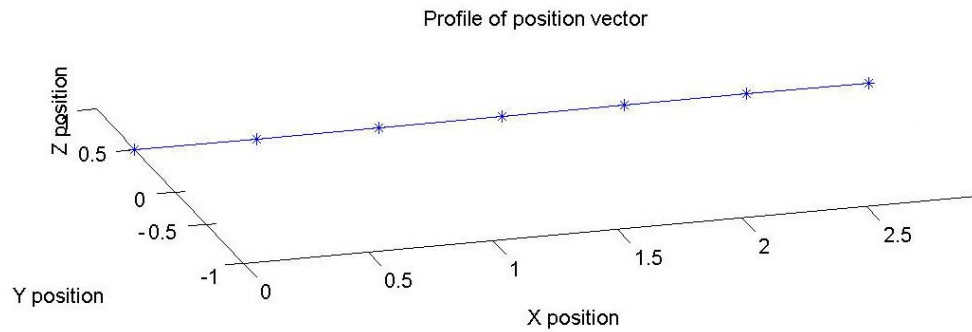


Figure 5.7: Line Test 2 Method 1--Abstract Model

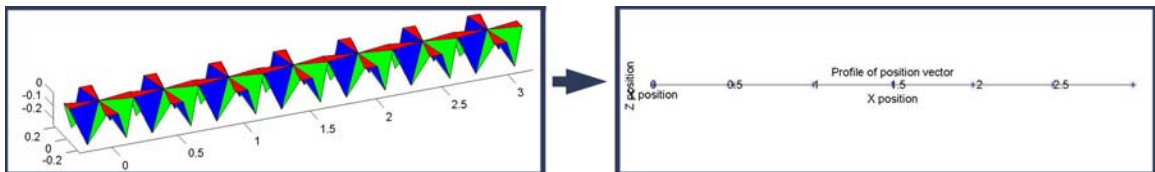


Figure 5.8: Line Test 2 Method 2-- With Fixed Face

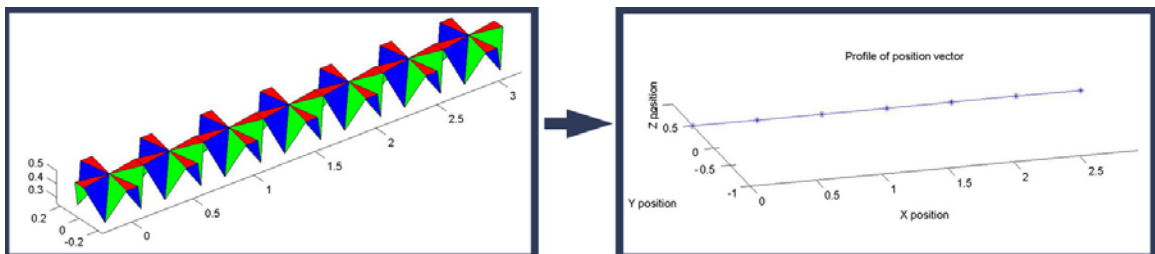


Figure 5.9: Line Test 2 Method 2-- Without Fixed Face

Again the kinematics table will introduce the comparison.

Table 5.6: Kinematics Comparison Test 2

	# Unknowns	# Equations	Degrees-of-freedom
Method 1	21	11	15
Method 2: Fix Face	63	1	66
Method 2:Free Face	66	1	69

The values do not change, because the new inputs only change the magnitude of the existing constraints.

Taking a quick glance says every graphs look the same. Then we look a little bit closer at Figure 5.8. The Z-values of the center-points are all at zero. The other graphs from the other programs show that the center-points are at the expected 0.5 unit. There is a difference. Let's look at the actual values by comparing the results in tables.

Table 5.7: Time Comparison for Line Test 2

	Time
Method 1	0.340 sec
Method 2: Fix Face	4.761 sec
Method 2:Free Face	5.421 sec

Again the first method is the fastest of all three. In this case Method 2: Fix Face and Method 2: Free Face switch ranking. A possible reasoning is that there are less unknown variables when one of the faces is fixed. So it will take less time when there are fewer variables. Also this is a straight horizontal line. It will not be difficult for any process to coverage to the answer.

The next table compares the number of iteration and the stored energy value.

Table 5.8: Iteration and Energy Comparison for Line Test 2

	Iteration	Energy
Method 1	10	4.850×10^{-6}
Method 2: Fix Face	1	7.470×10^{-9}
Method 2:Free Face	1	7.630×10^{-9}
Method 1- Method 2: Fix Face	9	4.840×10^{-6}
Method 1- Method 2: Fix Face	9	4.840×10^{-6}

Method 1 requires more iteration, but the convergence rate as seen in the previous table is faster. Both versions of Method 2 require the same number of iteration--1, but Method 2: Free Face has a slightly higher energy value. The difference is so small that it is insignificant.

The third table compares the Z-values as it was previous done for the first test.

Table 5.9: Z-values Comparison

Z-values Results	
M1 – M2:fix	M1 – M2:free
0.5	0
0.5	0
0.5	0
0.5	0
0.5	0
0.5	0
0.5	0

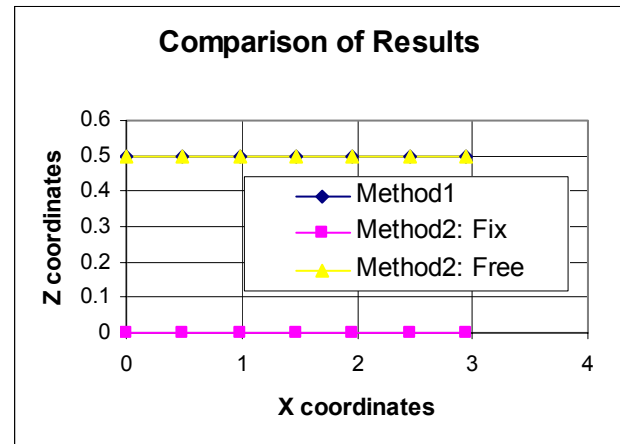


Figure 5.10: Z-Values Results for Line Test 2

From the previous table and graph, there is a recognizable difference. The result from Method 2: Fix Face converges to a line but all the z-values are at zero. The difference between Method 1 and Method 2: Fix Face is exactly the value of the inputs. Method 1 and Method 2: Free Face has the exact same value. Therefore their differences are zeros. Even in the graph, the resulted lines from Method 1 and Method 2: Free Face overlaps, leaving the graph to only look like it has two graphs. Of course the second line from the results of Method 2: Fix Face is strangely offset. Reason: the fixed face grounds the first face at Z-height of zero or at the edge of the future walls of the digital clay interactive device. This in turns causes the first unit cell to be grounded at (0,0,0). So how was face grounded? In the algorithm for Method 2, all the unknowns are placed into the iteration algorithm. The unknowns are the \bar{n} and \bar{v} values for every face on every unit cell of the matrix. To fix a face, we do not place the \bar{n} and \bar{v} values for that particular face into the iteration algorithm. This will prevent the iteration process from moving that face, which in turns prevents the first unit cell from moving. Since the first unit cell cannot move, the lowest energy state is when all the other cells are also at ground level. For Method 2 with all the faces being free, all the \bar{n} and \bar{v} values for every face are placed into the iteration algorithm. This allows the iteration algorithm to search for possible orientation of the \bar{n} and \bar{v} values for the all the faces.

5.4 Line Test 3

Now we notice there is a difference when the first unit cell of a 1-by-7 is displaced. With 3 inputs, the first at 0.3 unit, the middle at 0.1 unit, and the last at 0.6 unit, the results are similar to Line Test 2. Below are the resulting graphs.

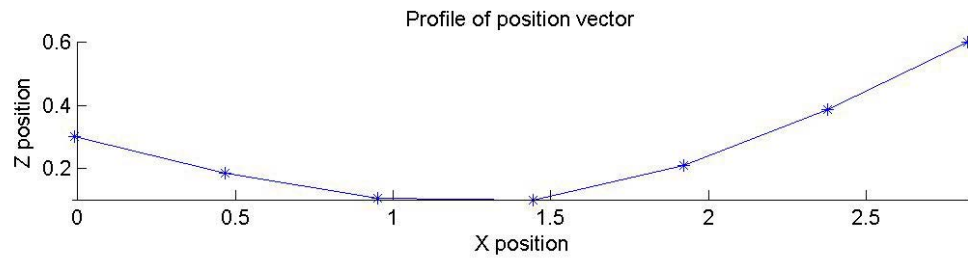


Figure 5.11: Line Test 2 Method 1-- Abstract Model

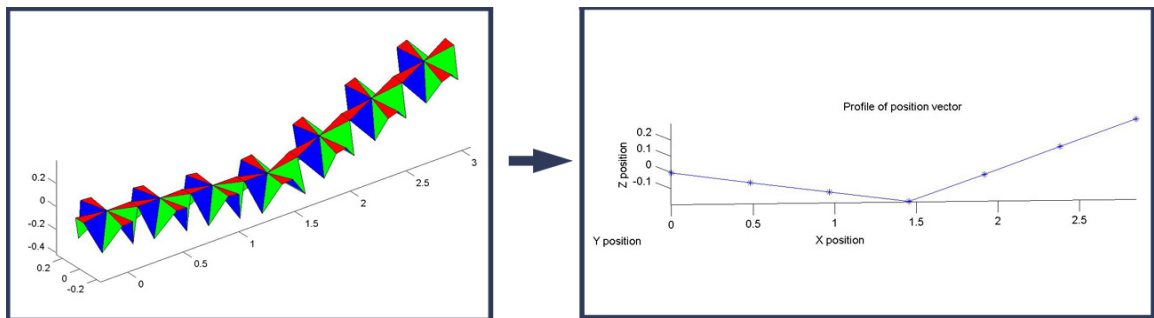


Figure 5.12: Line Test 2 Method 2--With Fixed Face

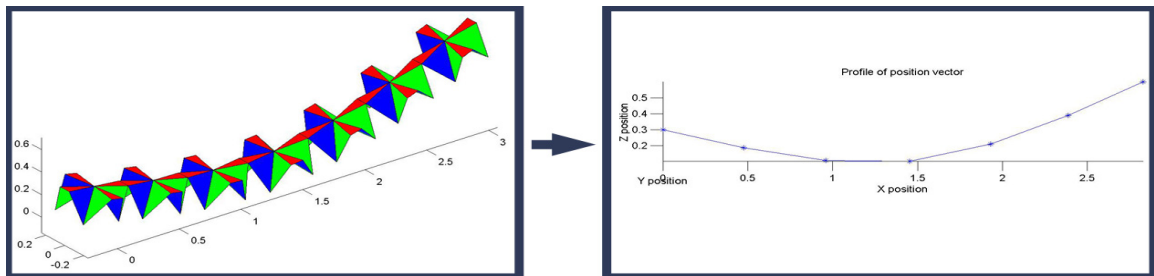


Figure 5.13: Line Test 2 Method 2--Without Fixed Face

Again the kinematics comparison chart is below.

Table 5.10: Kinematics Comparison Test 3

	# Unknowns	# Equations	Degrees-of-freedom
Method 1	21	12	15
Method 2: Fix Face	63	12+3=15	66
Method 2:Free Face	66	12	69

The only change from the first and second example is the number of equations for Method 1 because there is an additional input that constrained a different point than the other previous examples. The number of equations of both versions of Method 2 also changed because the constraints are different. The difference in the value was already explained in the previous chapter and will not be repeated here.

Obviously from the figures, the resulting graphs in Figures 5.11 look different from the other graphs. The Table 5.11 –5.13 below compare the exact values.

Table 5.11: Time Comparison for Line Test 3

	Time
Method 1	0.415 sec.
Method 2: Fix Face	212.354 sec ~ 3.539 min
Method 2:Free Face	239.330 sec~3.989 min

And again Method 1 is the winner with Method 2: Fix Face coming in on second. However the results from Method 2: Fix Face are questionable. Of course the results are based upon how the constraints are situated—this means whether the user wants to fix the edges to the bounding walls or not.

The next set of data is the iteration and energy comparison table.

Table 5.12: Iteration and Energy Comparison for Line Test 3

	Iteration	Energy
Method 1	10	27.90
Method 2: Fix Face	74	59.30
Method 2:Free Face	80	26.90
Method 1- Method 2: Fix Face	64	31.426
Method 1- Method 2: Free Face	70	0.971

In this configuration, the lowest/best energy value goes to Method 2: Free Face and Method 2: Fix Face has the highest/worst energy value. Again the difference between Method1 and Method 2: Free Face is not significant enough to be accounted for. The energy value for Method 2: Free Face is high because of the kink in the graph. Again the kink is to maintain the first edge face to be fixed horizontally to the ground, which in turns keeps the first unit cell being fixed at (0,0,0). However, the kink causes the resulted line to have a high energy value.

The next set of data compares the Z-values.

Table 5.13: Z-values Comparison

Z-values Results	
M1 – M2:fix	M1 – M2:free
0.3	0
0.2506	0.0013
0.2383	0.0016
0.2962	0.0008
0.2426	0.0014
0.254	0.0011
0.3011	0.0004

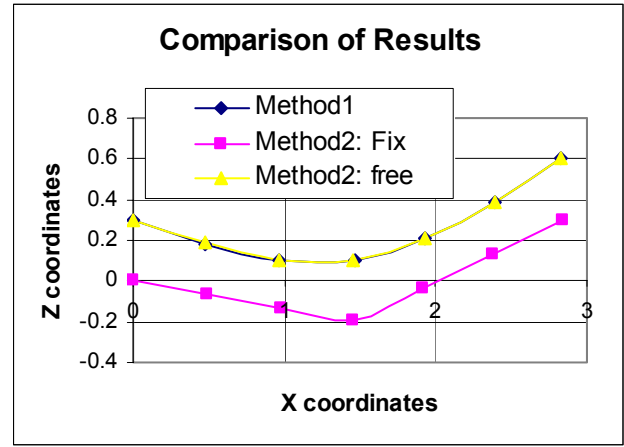


Figure 5.14: Z-Values Results for Line Test 3

From Table 5.13, there is a definite difference between Method 1 and 2. The difference is again noticeable in the corresponding graph where the results for Method 2: Fix Face is shown as the lower line. There is a slight difference between Method 1 and Method 2: Free Face. The difference is so little that the two lines overlap in the corresponding graph. These results conclude that Method 2: Fix Face cannot properly handle an input at the first unit cell. Previously there was a section discussing how Method 2: Fix Face handles non-first-unit-cell inputs. Later there will be more examples of Method 2: Fix Face handles non-first-unit-cell inputs.

Unlike the first two examples, we are going to introduce a new comparison chart: comparing the angles. Yes, any of the programs that implement the methods can output the angles of the joints as well as the center-point positions. For every unit cell, there are 12 angles. In a matrix of 1-by-7 there will be a total of $12 \times 7 = 84$ angles. To properly

compare the results from method 1 and Method 2, we need to show 84×3 angles = 252 plus the comparison values. We are not going to show all of that in this chapter. Please refer to Appendix B for all the values and the comparison values for Line Test 3. Table 5.14- 5.16 will show a sample of angles. Note that the angles are symmetric about the line that connects two center-points together. Refer to the previous Figure 5.12 and 5.13 for visual help.

The legend for the tables below:

- After: Joint angles (radian) after the program converges
- Before: Joint angles (radian) before the program converges
- Diff: the absolute value difference between the before and after angles
- Energy: the energy stored in the spring
 - Method 1: $k \cdot (\text{after} - \text{before})^2$. K: Average Stiffness Value: 0.0756 Nm
 - Method 2:
 - Energy Equation for larger joint: $0.1348 \cdot (\text{Before} - \text{After})^2$
 - Energy Equation for larger joint: $0.0164 \cdot (\text{Before} - \text{After})^2$
- The alphabet letters: the symmetry of the angles:
 - For example:
 - All A values for the Cell 1 has similar values in Method 1.

Table 5.14: Joint Results for Method 1

	Cell 4			
	After	Before	Diff	Energy
A	2.2246	2.0089	0.2157	0.007072
B	1.4079	1.4432	0.0353	1.93E-05
C	2.1227	2.0089	0.1138	0.001968
C	2.0378	2.0089	0.0289	1.29E-05
B	1.4295	1.4432	0.0137	2.85E-05
A	2.2151	2.0089	0.2062	0.000659
D	2.2272	2.0089	0.2183	0.007244
E	1.4068	1.4432	0.0364	2.05E-05
F	2.0997	2.0089	0.0908	0.001253
F	2.0514	2.0089	0.0425	2.8E-05
E	1.4126	1.4432	0.0306	0.000142
D	2.2104	2.0089	0.2015	0.000629

Table 5.15: Joint Results for Method 2 Fix

	Cell 4			
	After	Before	Diff	Energy
A	2.2246	2.0089	0.2157	0.007072
B	1.4079	1.4432	0.0353	1.93E-05
C	2.1227	2.0089	0.1138	0.001968
C	2.0378	2.0089	0.0289	1.29E-05
B	1.4295	1.4432	0.0137	2.85E-05
A	2.2151	2.0089	0.2062	0.000659
D	2.2272	2.0089	0.2183	0.007244
E	1.4068	1.4432	0.0364	2.05E-05
F	2.0997	2.0089	0.0908	0.001253
F	2.0514	2.0089	0.0425	2.8E-05
E	1.4126	1.4432	0.0306	0.000142
D	2.2104	2.0089	0.2015	0.000629

Table 5.16: Joint Results for Method 2 Free

	Cell 4			
	After	Before	Diff	Energy
A	2.1199	2.0089	0.111	0.001873
B	1.4369	1.4432	0.0063	6.15E-07
C	2.0377	2.0089	0.0288	0.000126
C	2.0373	2.0089	0.0284	1.25E-05
B	1.4373	1.4432	0.0059	5.29E-06
A	2.1202	2.0089	0.1113	0.000192
D	2.1202	2.0089	0.1113	0.001883
E	1.4373	1.4432	0.0059	5.4E-07
F	2.0372	2.0089	0.0283	0.000122
F	2.0378	2.0089	0.0289	1.29E-05
E	1.4368	1.4432	0.0064	6.23E-06
D	2.1198	2.0089	0.1109	0.000191

From the tables, the symmetry becomes apparent. Each cell is divided into two halves. Within each half are other symmetry among the joints. A-C denotes the symmetry for the first half. D-F denotes the symmetry among the second half. The symmetry is repeated in every column. This in turn causes the energy in the springs to be symmetric as well. From the symmetry, we can say that joint angles are correctly calculated.

Next, we will be talking about how to improve the speed and accuracy of the programs.

5.5 Line Test 4: Piggybacking Style

From all previous graphs of Line Test 1, 2, and 3, Method 1 converges the fastest to highly reasonable answers that are compared to Method 2: Free Face. Unlike Method 2: Fix Face where the edge of the face for the first unit cell is fixed, both Method 1 and Method 2: Free Face have similar constraints. Although Method 1 converges the fastest, Method 2: Free Face is a more accurate representation of the manufacturable crust that was described in the earlier chapters. We can take advantage of this: use the results from Method 1 as the inputs for Method 2: Free Face. Unlike Method 1, Method 2: Free Face does not restrict the algorithm from searching around the inputs for other possible combinations that might create a lower potential energy state. Also this will decrease the time in which Method 2: Free Face converges to an answer because the initial guess (or in this case the inputs) is a lot closer to the answers.

Because Method 2: Fix Face has a face constraint that is different from any of the other programs, Method 2: Fix Face will not be piggybacked.

To demonstrate this piggyback style, we will use two examples. The first one is from Line Test 3 with three inputs for a 1-by-7 line. The second example is a 1-by-31 line with 5 inputs.

5.5.1 Piggybacking Style Example 1

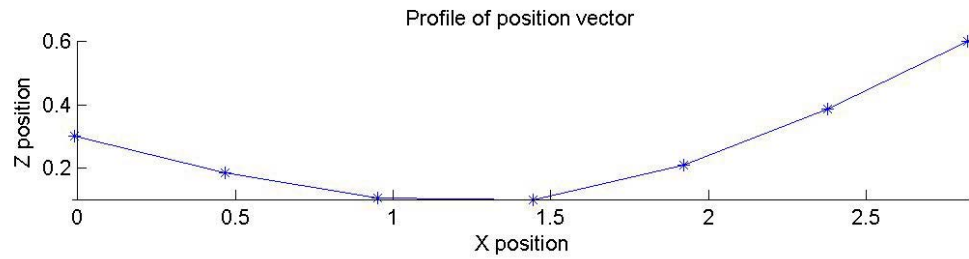


Figure 5.15: Graphic Reminder of Previous Results for Method 1 and 2

Because the results for Both Method 1 and Method 2: Free Face are so similar, it is unnecessary to show two graphs that look identical. For this case, one graph is enough information to convey the point.

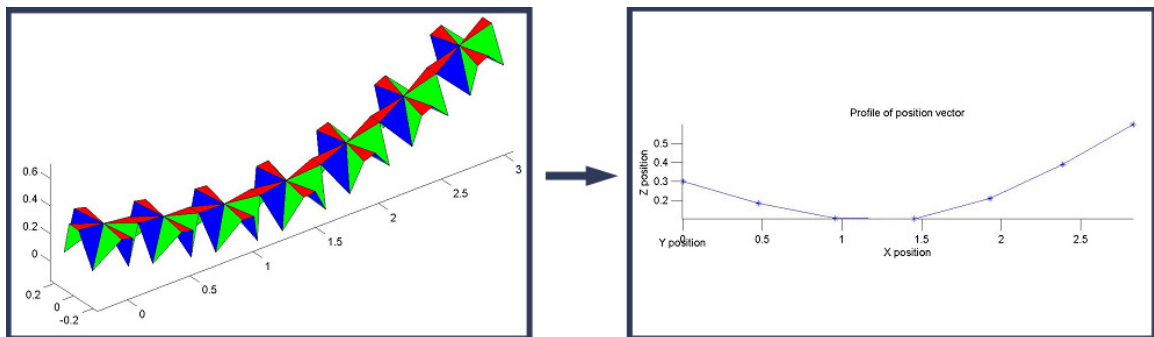


Figure 5.16: Piggybacked Example 1 Result

The piggybacked result looks indistinguishably the same. However, let's look at the details by using tables to actually compare the difference.

Table 5.17: Time Comparison for Piggyback Example 1

	Time
Method 1	0.415 sec.
Method 2:Free Face	239 sec. ~ 4 min.
Piggyback: M1+M2 Free	215 sec. ~3.6 min.

Piggybacking results shows that the summation of the total time is 3.6 minutes. The total time is the time it takes for Method 1 to converge then summing it up with the time it takes to run the results through Method 2: Free Face and arrive at the final answer. It appears that piggybacking for this example is faster by 0.4 minute than just running Method 2: Free Face.

Table 5.18: Iteration and Energy Comparison for Piggyback Example 1

	Iteration	Energy
Method 1	10	4.850×10^{-6}
Method 2:Free Face	80	26.90
PG:Method 1+ Method 2:free	70	26.97
 Method 1- Method 2: Free Face 	9	26.905
 Method 1- P G 	9	26.917

Even comparing with Table 5.18, The Piggybacking has better results than just applying Method 2: Free Face. The Piggybacking method takes less iteration and the energy value is almost the same- minor difference.

Table 5.19: Z-values Comparison

Z-values Results	
M1 – M2:free	M1 – Piggyback
0	0.0
0.0013	0.0
0.0016	0.0
0.0008	0.0007
0.0014	$1.0 * 10^{-4}$
0.0011	0
0.0004	0.0004

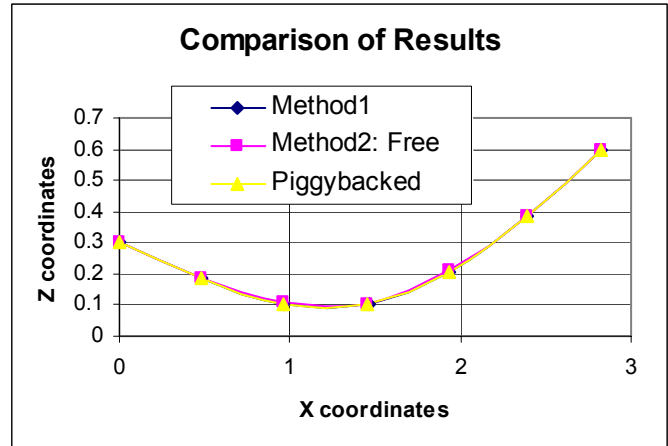


Figure 5.17: Z-Values for Piggyback Example 1

From applying the piggyback method, the differences in the Z-values are very small--almost all zero. In the corresponding graph, all the lines overlap giving an illusion of one line. Now the question is: is piggy-backing worth it? The difference is so small that it may not be significant enough to account for in our prediction. However, this is a simple case. Let's consider a more complex case with 1-by-31 unit cells.

5.5.2 Piggybacking Style Example 2

This example will use a 1-by-31 line with five inputs. Below are the results for applying the Method 1, Method 2: Free Face, and finally the results from piggybacking Method 1 onto Method 2: Free Face.

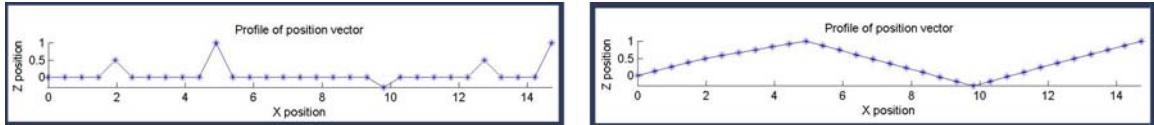


Figure 5.18: Method 1 Inputs (Left) and Line Interpolation (Right)

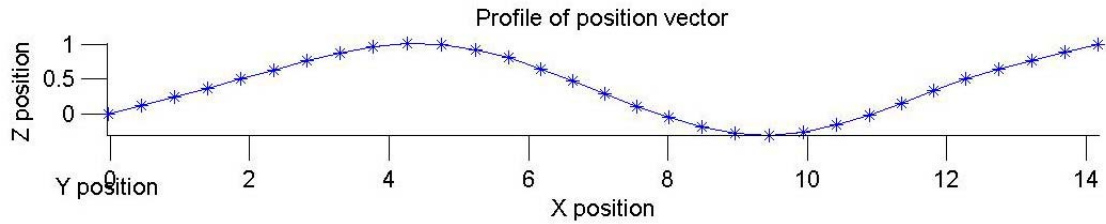


Figure 5.19: Method 1 Results

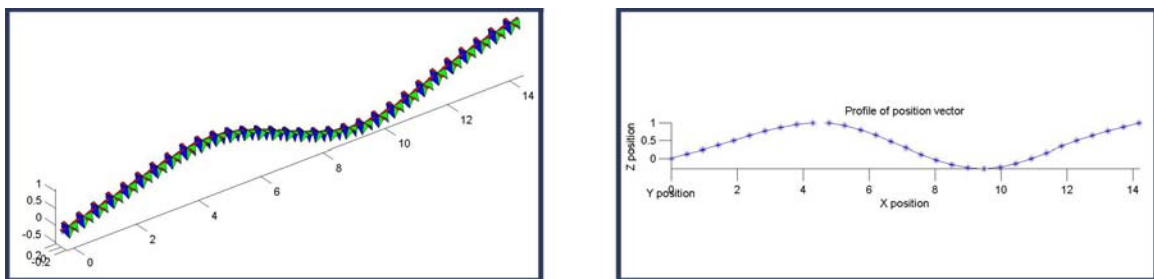


Figure 5.20: Piggyback Results

Again, the Piggyback results look similar to the Method 1 results, because Method 1 has proven to be very reliable on the results as seen in previous line tests and example. Below are the tables that describe in more detail of the final answers.

Table 5.20: Time Comparison for Piggyback Example 2

	Time
Method 1	1.906 sec.
Method 2:Free Face	1990 sec. ~ 5.5 hrs
Piggyback: M1+M2 Free	1470 sec. ~ 4.1 hrs

As the number of unit cells increases, the time it takes for each program to converge to an answer also increases. Instead of minutes, now Method 2: Free Face takes 5.5 hours, which is 5 hours and 30 minutes. With piggybacking, it takes 4.1 hours, 1.4 hour less than without Method 2: Free Face being piggybacked. Of course, just running Method 1, it only takes less than 2 second.

The next table compares the iteration and energy values.

Table 5.21: Iteration and Energy Comparison for Piggyback Example 2

	Iteration	Energy
Method 1	32	34.859
Method 2:Free Face	283	34.684
PG:Method 1+ Method 2:free	295	34.703
 Method 1- Method 2: Free Face 	251	0.1749
 Method 1- P G 	263	0.1547

Using the piggybacked method, the number of iterations does increase to 295 iterations in comparison to 283 using just Method 2: Free Face without piggybacking. Although the iteration numbers differ, all energy values are very similar. It is insignificant to distinguish among the differences in values, refer to the set of data below.

Table 5.22: Z- values Comparison

Z-values Results	
M1 – M2:free	M1 – Piggyback
0	0
-0.0002	0
-0.0002	0
-0.0002	0
$-1*10^{-4}$	$-1*10^{-4}$
0	0
0.0003	0
0.0007	0
0.001	0
0.001	0
0.0003	0.0003
-0.0014	0
-0.0032	0
-0.0049	0
-0.0058	0
-0.006	0
-0.0053	0
-0.004	0
-0.0024	0
-0.0009	0
-0.0003	-0.0003
-0.0011	0
-0.0019	0
-0.002	0
-0.0016	0
-0.0007	0
0.0002	0.0002
0.0006	0
0.0006	0
0.0004	0
$-1*10^{-4}$	$-1*10^{-4}$

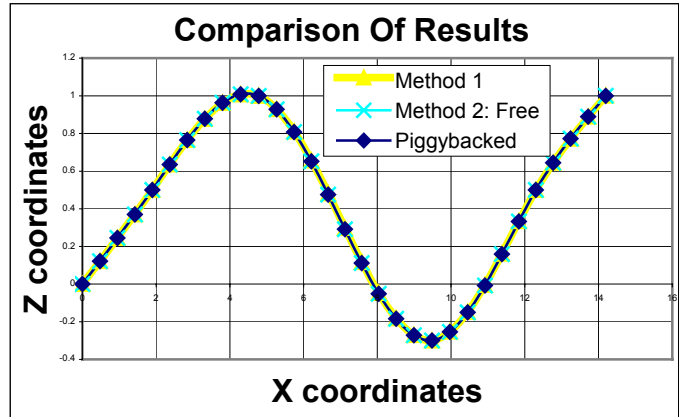


Figure 5.21: Z-Values for Piggyback Example 2

From Table 5.22, the actual number comparison shows there really isn't much difference between running Method 1, Method 2: Free Face, and piggybacking them. So what is the benefit for running a program that takes hours and the answers will be very similar to another program that take several seconds to complete the same task?

The answer: how accurate do you want your answer to be? Six Sigma accurate? For the Digital Clay project, I do not think it is that necessary to be that perfect at this stage of the design process. Therefore, the rest of this chapter will switch back to focusing on the methods not the program. Method 1: Abstract Model of Crust. The next section will compare how Method 1 handles different size matrix. It does not matter about the dimension as much as the number of unit cells and inputs in the matrix. The number of unit cells determines the number of unknowns. The more inputs, the more equations, this would decrease the computational time because there are more constraints that would guide the iteration process. The more numbers of unknowns, the longer it takes for the methods to converge to an answer because there are more variables to iterate. We will not test these two assumptions because the location and the magnitude of the inputs vary with the user and the situation. Any resulting data from these test will be misleading.

5.6 Method 1 Matrix Elapse Time Study

From previous tests, we learned that the program that implements Method 1 converges to the answer the fastest of any of the other programs. So how will it handle different size matrices as the number of unit cells increase? To answer this question, we will only give each matrix one input of the same value at the center or around the center of the matrix. The stiffness value will be arbitrarily assigned at 1000 units. The corners of the matrix will default to the original constraints of Method 1 at zero Z-heights. Figures 5.21-5.23 are examples of how we are conducting this experiment.

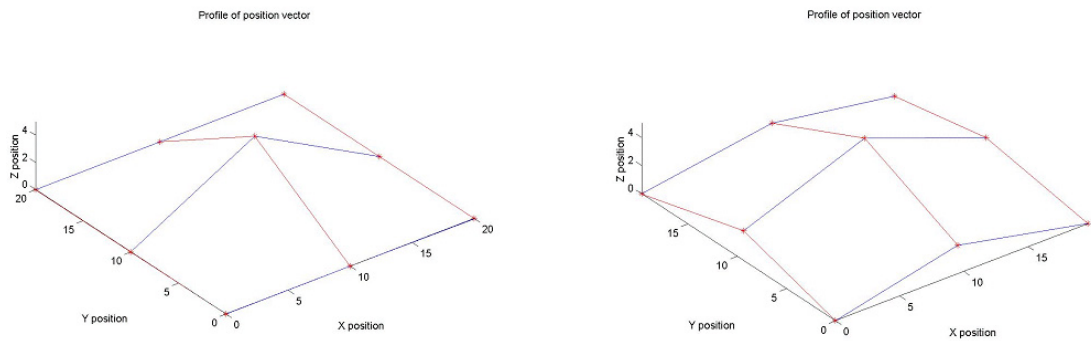


Figure 5.22: 3-by-3 Matrix Input at [2,2] (Left). Results (Right)

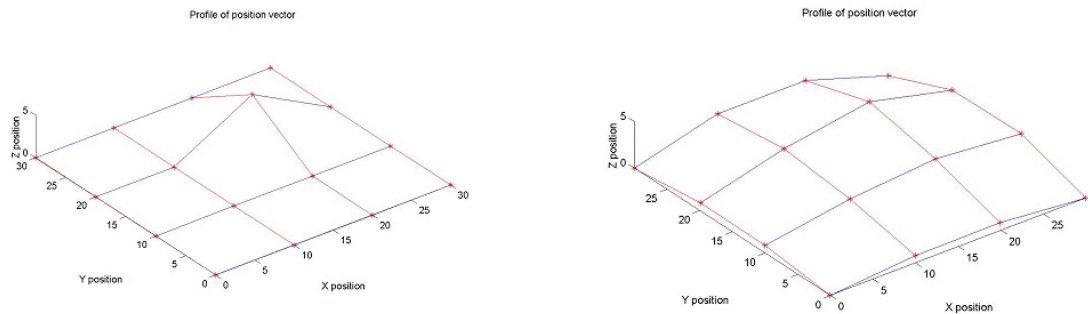


Figure 5.23: 4-by-4 Matrix Input at [3,3] (Left). Results (Right)

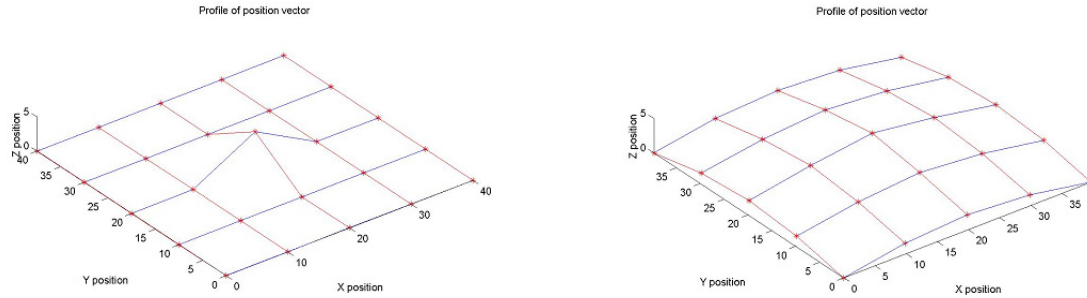


Figure 5.24: 5-by-5 Matrix Input at [3,3] (Left). Results (Right)

We continue performing this test until the size of the matrix is 15-by-15. Figure 5.25 and 5.25 are two graphs from the test.

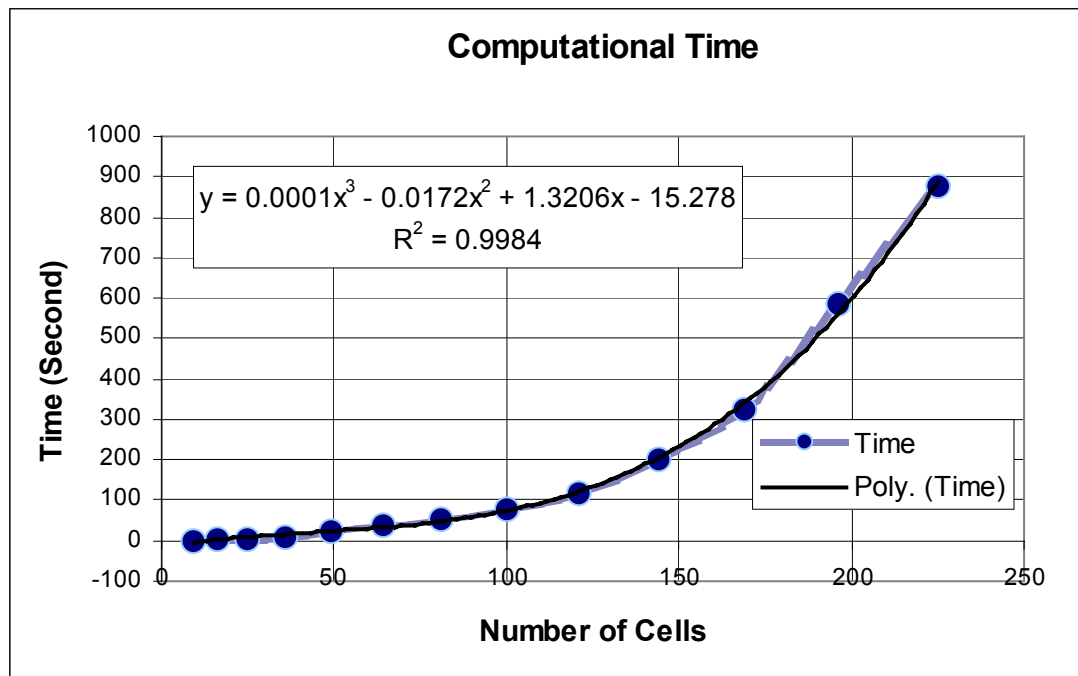


Figure 5.25: Computational Time

Figure 5.26 shows that as the number of cells increases, the computational time increases. The graph follows a 3rd degree polynomial line.

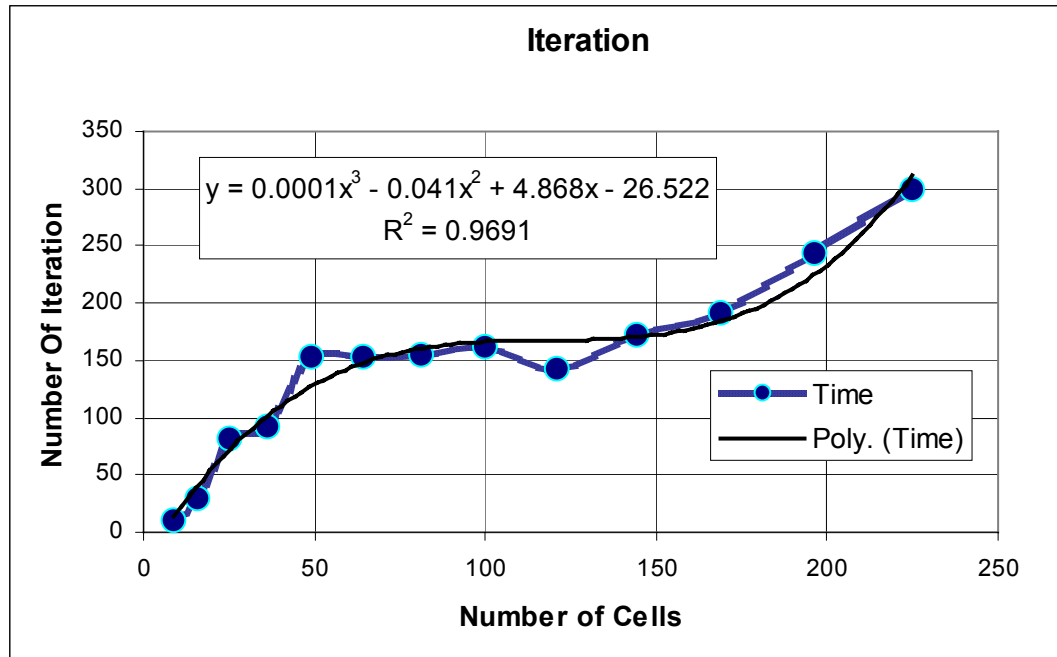


Figure 5.26: Iteration Rate

The number of iterations as seen in Figure 5.26 also follows a rising 3rd degree polynomial line as the number of cells increases.

From these graphs, we can predict the time it takes and the numbers of iterations for Method 1 to converge to an answer when it is applied to a matrix of any size.

5.7 Method 1 Matrix Accuracy Prediction

At this point of time, the mechanical device that will deform the crust has not been built yet. Therefore, Method 1 cannot be proven with experimental data to predict the deformation of the crust based upon the user inputs. The current car-hood SLA model is not a precise hardware that will displace any center-points by any specify distant. However, visual inspection and intuition can be applied to test Method 1 for reasonable answers. Below are several tests using a 6-by-4 matrix. The dimensions are varied from the previous matrix size to show that this method can handle various conditions.

5.7.1 Matrix Plane Test 1

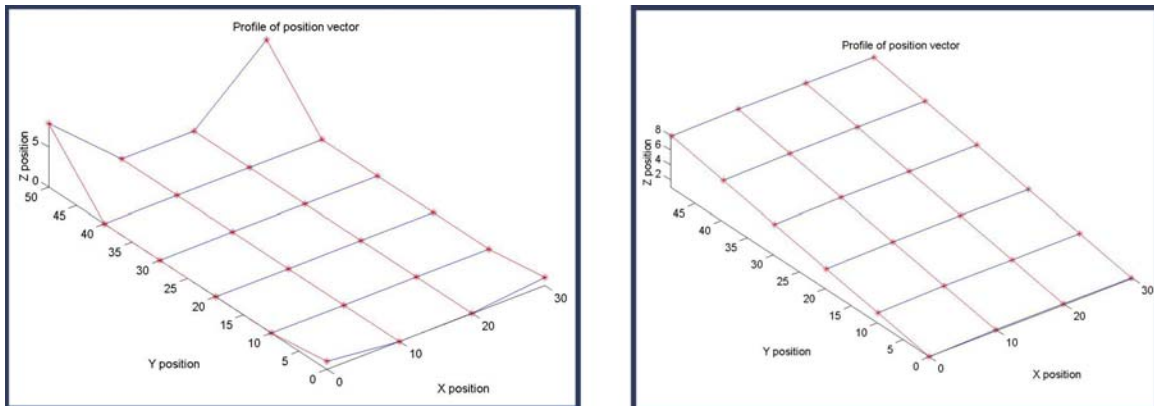


Figure 5.27: Plane Test Inputs (Left). Results (Right)

From intuition, if two corners are vertically displaced for a material with high stiffness of 1000 units, the results should be a plane. For this case, the result is a plane. A real-life example is a piece of paper fixed at one end and held by two fingers at the other end—the result is a plane.

5.7.2 Matrix Surface Test 2

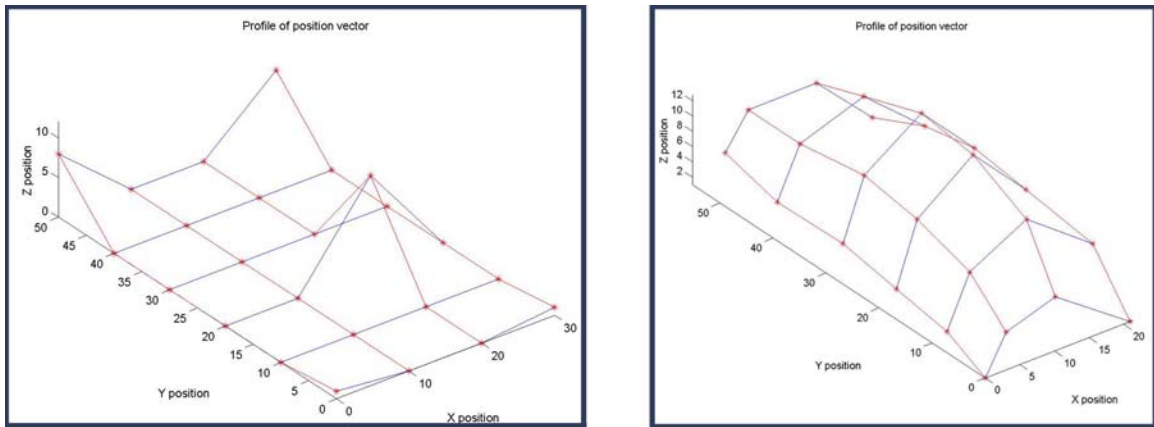


Figure 5.28: Surface Test Inputs (Left). Results (Right)

If a displacement is applied to point in the middle of a plane, the plane should curve and create a concave surface. This example is again compared to a real-life situation with a piece of paper. If a piece of paper is held in space with one finger in the middle, how will it look like? The result looks like the graph on the right.

We already did several line tests. From these results, we concluded that Method 1 does a very good job at predicting the deformation. Now we can apply this method to the car hood models.

5.8 Mimicking the Car-hood Models

In previous chapters, we use the car-hood model as an example of what we wanted the low degree of freedom machine to mimic. Below is a repeated image of the car-hood models as they morph.

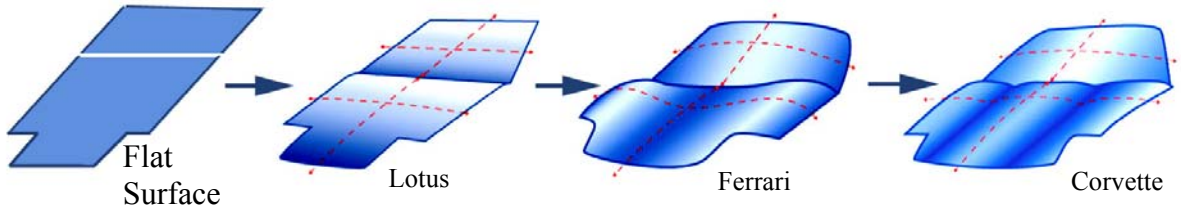


Figure 5.29: Morphing Car-hood Models

Now we want to mimic the car-hood using Method 1. Only the hood will be modeled. The windshield needs another matrix, because it is on a different plane than the hood. The Lotus will not be attempted, because it is just a flat plane. That plane can be seen in Figure 5.27.

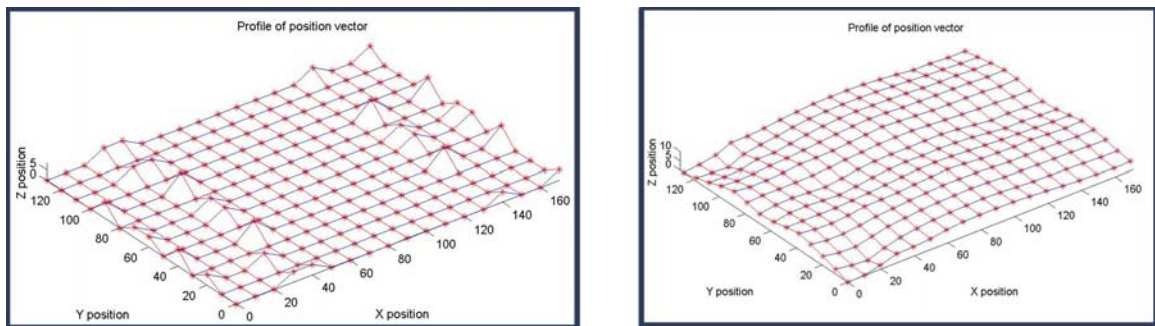


Figure 5.30: Ferrari Attempt

This is the first attempt with 30 inputs on a 14-by-18 matrix. It took 13 hours for the program to finally converge to an answer. The mimicking hood is close, but not

perfect. Figure 5.30 demonstrates the possibility of recreating the hood and to not show how perfect the mimicking is. We only did one run. We can run various trials and errors by changing the magnitude of the inputs and the amount of inputs until Method 1 shows a perfect Ferrari. However, the benefit will not be worth the time and effort put into this experiment.

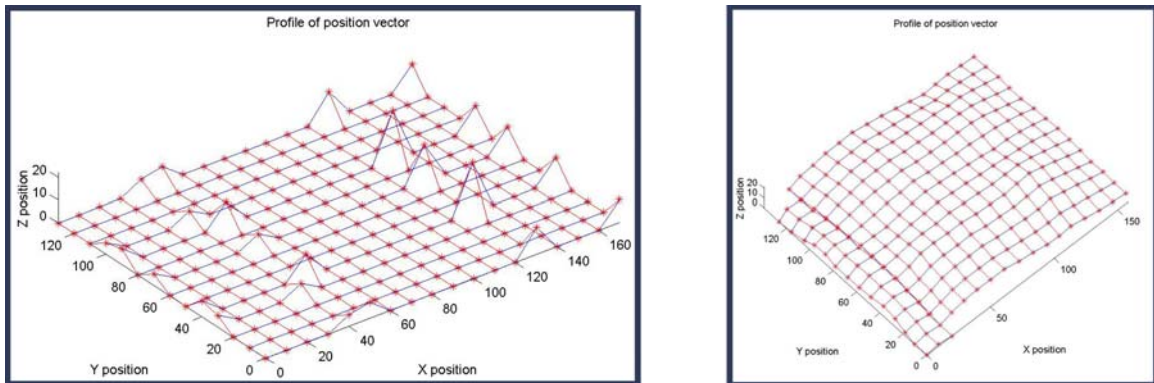


Figure 5.31: Corvette Results

Figure 5.31 contains 32 inputs for 14-by-18 matrix to model the Corvette hood. The time for the answer to converge is 15 hours. The results (image on the right) show a kink near the bottom left hand corner because there are several inputs that are right next to each other. These inputs cause a sharp decline from one surface to the other. Again, we can run several experiments until the perfect hood is displayed, but the benefit will not be high enough to compensate for the resources added into getting the perfect hood example.

5.9 Degeneracy

Degeneracy is when a system would reduce or degenerate to a simpler version of itself. For an example a dot is a degenerate version of a circle with radius 0. A circle is a degenerate version of an ellipse with eccentricity 0. The system of rods and springs in Method 1 can degenerate when the matrix is deformed horizontally. In other words, we did not consider the shearing effect of the matrix because there isn't any spring between two intersecting rods. Figure 5.32 shows where the springs would be to prevent shearing in addition to the existing springs underneath the center-points.

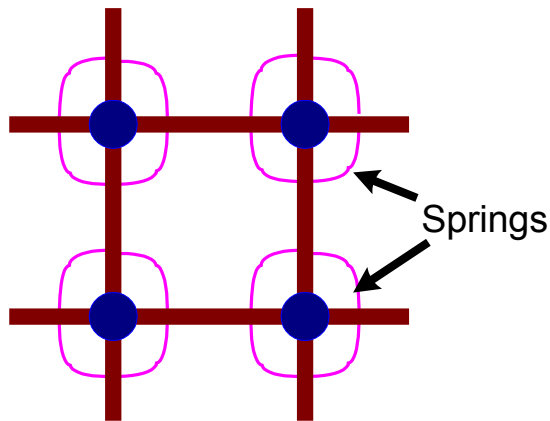


Figure 5.32: Non-Existing Springs

Previously we stated that if various Z-heights are inputted, the shape deformation for the matrix in Method 1 can be predicted from minimizing the energy in the angular springs. However, if the matrix is sheared as seen in the right image of Figure 5.33, the Potential Energy calculated will be the same as the undeformed state, as shown in the left image.

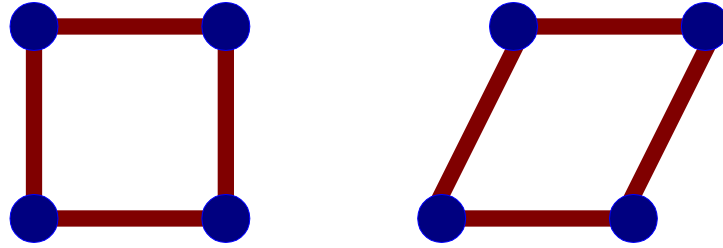


Figure 5.33: Non-deformed state (Left) Shear Deformation (Right)

This would make Method 1 solution NOT unique when deforming horizontally. This means that both images in Figure 5.33 have the same amount of stored energy but their shapes are different. Uniqueness will be further explained in Chapter 5. To resolve this situation, we need to place a spring between any two intersecting rods as seen in Figure 5.32.

For Method 2, there are two types of springs that comprise one unit cell. Any type of deformation of the unit cell, including shear, will change the total energy in the springs. Therefore, this spring configuration will prevent Method 2 from degenerating when deformation is horizontally applied. The potential energy is different in the two cases in Figure 5.33.

5.10 Uniqueness

Uniqueness is when the methods would converge to the same results, based upon the given inputs and constraints and given different initial guesses. As previously stated, since Method 1 can degenerate when deforming horizontally, Method 1 is not unique in that deformation manner. Method 2 has two types of springs that may prevent

degeneration and increase the possibility of uniqueness. However Figure 5.34 is an example of horizontal deformation that questions if uniqueness still exists for either method. When one edge of the matrix is constrained and a force is applied to the other edge as seen in the first image of Figure 5.34, what will the shape be?

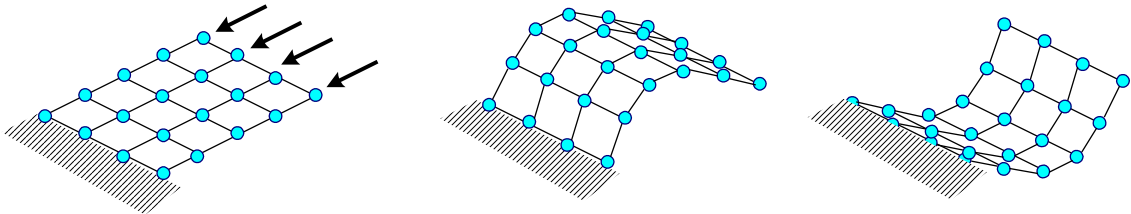


Figure 5.34: The Bowing Effects

Will it bow up as seen in the middle image of Figure 5.34 or bow down as seen in the last image? Or can it be a combination of both where half of the matrix bows up and the other half bows down? Both the bowing up and bowing down situations are minima, and the results are based upon the searching process: how and which direction the guesses would take to search for the answer. Therefore the answer is NO: neither method is unique for horizontal displacement. A possible solution for guiding the results is applying threshold values to ensure that any joints angles will not be greater than a certain amount of degrees. Threshold values will be explained in Chapter 6.

For Method 2, the springs are located where the actual joints are. This placement will guide Method 2 into behaving more like the actual model. However, it is difficult to guarantee that the physical model will behave in the same manner as the math model. Another possible solution to get the desired effect is adding a Z-height displacement in

the center of the matrix to ensure that the matrix would bow up. The more Z-heights (or constraints added), the more likely the system would converge to a unique answer. The same solution can be applied to ensure a bowing down effect.

Another question is: are the methods unique for vertical deformation, (ie. when Z-heights are used as inputs)? One test for the vertical deformation situation is performing a set of simulation runs. The runs include comparing the results from one run with the results from another run using different starting positions but with the same constraints as seen in Figure 5.35-5.37 with Method 1.

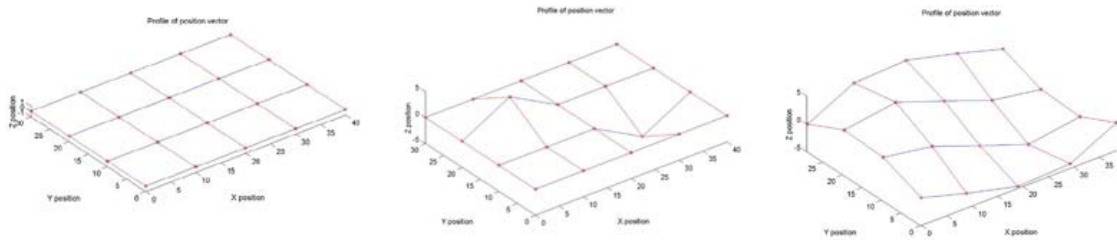


Figure 5.35: Set 1. Flat Plane Starting Position (LT). Input (M). Results (RT).

Set 1 is a 4-by-5 example that starts off with a flat plane as seen in Figure 5.35. Then two Z-heights are inputted as seen in the middle image. The results as expected are seen in the last image.

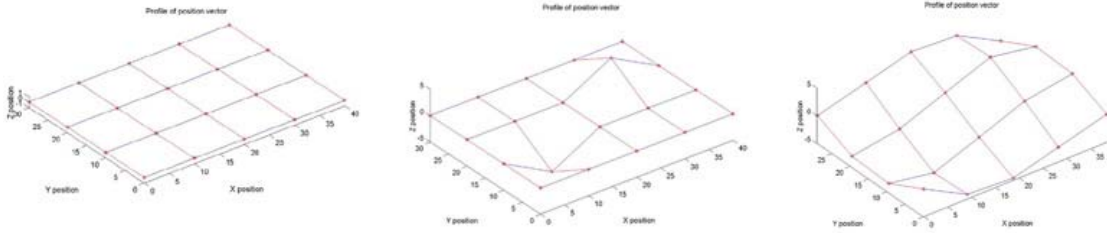


Figure 5.36: Set 2. Flat Plane Starting Position (LT). Input (M). Results (RT).

Similarly Set 2 also starts off with a flat plane as seen in Figure 5.36. The Z-height inputs are placed in the middle image and the results are seen in the last image.

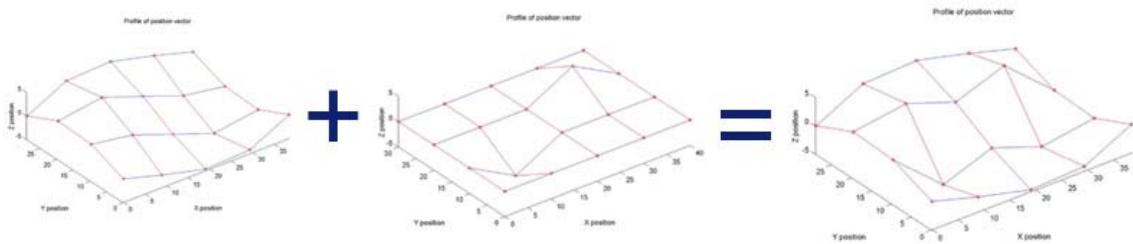


Figure 5.37: Set 3 Starting Position with Inputs.

Using the results from Set 1 as seen in the first image of Figure 5.37 and adding to it the Z-heights inputs from Set 2 as seen in the middle, a new starting position will be specified that should converge to the same position as in set 2, seen in the last image of Figure 5.37.

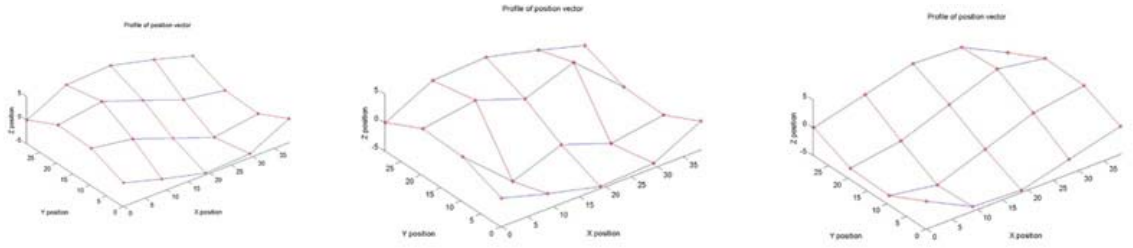


Figure 5.38: Set 3. Starting Position (LT). Input (M). Results (RT).

For Set 3 the starting position (results from Set 1) is seen in the first image in Figure 5.38. The Z-heights from Set 2 are added into the starting position as seen in the middle image. And the results look exactly like Set 2, as one would expect. When the data points are compared, the difference is $1/10000$ in the data point values. This difference is insignificant. For this example, the implementation of Method 1 is unique. Since different starting points converge to the same solution, the question is: for Method 1 are the solutions always unique? The answer is unknown. Although the previous example shown indicate that Method 1 is unique and most likely there would be a lot of cases where the results will be unique, it is difficult to determine how the iterative process would converge for any conceivable inputs.

Similar to Method 1 uniqueness test, Method 2 can also go through various tests using a serial chain of unit cells instead of matrices. Of course, when Method 2 has been implemented for a matrix, the 4-by-5 matrix test can be repeated. Again the solutions for Method 2 are not likely to be unique due to the iterative guessing process for determining the results.

5.11 Geometric Non-linearity

Geometric non-linearity could occur when there is a large displacement in relation to the matrix dimension. In the case of the structures discussed in this thesis, the both the X-Y position coordinates of the crust are fixed only at the first point. Since the other points of the crust can translate in the X or Y or both in response to the Z-displacement; stretching-induced geometric non-linearity are partially mitigated. However, in more complex models of translation-restricted clay, geometric non-linearity may have to be taken into account. Geometric non-linearity will create unwanted stress stiffening and/or strain deformation either in the joints or the rigid links of the matrix. This will cause the joints or the links to warp undesirably. In relation to the Digital Clay crust matrix, understanding geometric non-linearity can help determine how much Z-height displacement amount a matrix can handle before an unwanted amount of stress or strain would occur. By knowing this value, we can prevent the structural damage of the crust matrix or the machine that deforms the crust. There are two approaches for handling Geometric Nonlinearity: Updated Lagrangian description and Total Lagrangian description (El-Zeiny, 2000) (Bathe, 1996). Update Lagrangian refers to comparing the deformation geometry to a previous value. Total Lagrangian refers to comparing the deformation to a reference configuration. One way to implement either one of the Lagrangian descriptions is applying Finite Element Analysis (FEA) individually to every situation. This application was already attempted by He Liu and co-workers to determine the geometric nonlinearity occurrence of a Fluid Tank, which will help determine the effect of large deformation to the tank. (Liu, et. al, 2002). A similar process can be extrapolated to the matrix for either method. Currently the FEA implementation has not

been performed, but it is assumed that there is a limit to how much deformation the matrix can handle before the matrix warps. Geometric Non-linearity can be a supplementary evaluation with using a threshold value. Threshold value will be mentioned in Chapter 6.

5.12 Ending Remarks

From the results shown in this chapter, the methods developed can predict the deformation based upon the user inputs and the material stiffness property. There are questions with uniqueness and geometric non-linearity that was already discussed.

5.13 Comparing Two Theses

Previously Paul Bosscher's Master Thesis was referenced. Bosscher is a former graduate student who was involved in the Digital Clay project. For his thesis, he helped designed the kinematics structure of the deformable crust and developed an algorithm to predict the deformation of the crust. The algorithm is for an abstract matrix with angular springs between vertices and linear springs representing prismatic joints as shown in Figure 5.39.

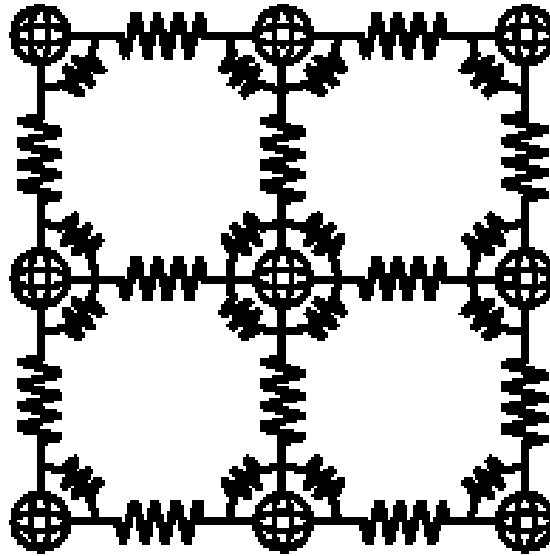


Figure 5.39: Bosscher's Abstract model

Comparing Figure 5.39 to Method 1's abstract model in Figure 4.9 there are two obvious differences; the locations of the springs that are being minimized and the use of prismatic joints. For Method 1 from this thesis, prismatic joints are not modeled. However, the implementation of Method 1 can be modified to consider the prismatic joints by changing the tolerance value for the length calculation.

Another main difference is the minimization technique. Bosscher finds the minimal location of all the points by moving a set of points while fixing the other sets of points. This method did yield the minimal location of all the points. For this thesis, we move all the points at the same time searching for various combinations of the location of the center-points. It is uncertain which style of problem solving is better: solving for one big equation or lots of smaller equations.

For Method 2, unlike Bosscher's approach, the actual manufactured crust shape was modeled. This means there are two different angular springs. Each spring represents the different joints in the unit cells. This is not an abstract model, which increases the complexity. Because the crust is fully modeled, the location of every vertex can be calculating using Method 2 before and after deformation. In addition the edge vertices can be calculated as shown in Figure 4.44.

Another main difference between this work and Bosscher's thesis is the application of material properties: the stiffness values of the joints are derived from experimental runs. Knowing these stiffness values will enable us to accurately predict the deformation capability of the material being used. Bosscher did consider the material properties, stating the springs have a stiffness value, but did not develop a process for deriving what these values are. For both methods from this thesis, the actual stiffness values were used to predict the deformation of the matrix.

CHAPTER 6

LAST COMMENTS

In this chapter, the “Concluding Comments” will be discussed first then followed by “Future Works” well as the “Benefits and Values” for each individual subgroup in the Digital Clay team. There are two methods developed for this thesis with three different programs. The suggested method to use is Method 1. Of course it is up to the end user which method and which program best fit the usage.

6.1 Concluding Comments

At the beginning we attempt on fulfilling four goals and answering the key question: **What is this kinematics structure and how can the deformation of the kinematics structure be predicted based upon the materials being used and constraints being applied?** Below is a list that comments on the accomplishment.

1. Designing and manufacturing a deformable kinematics structure

The final design is a deformable crust matrix with spherical joints. The crust design can be formed into various shapes. The design is effective but of course there can be improvements in simplifying the design to better scale down the matrix. For manufacturing the crust, several manufacturing techniques are listed. Some of them show potential in manufacturing the crust matrix. We settled on the SLA process because it was the most accessible and the most time efficient. The one of the criteria for selecting the process for manufacturing the kinematics structure is obtaining the stiffness property

from the material used in the chosen process. The stiffness factor will determine the deformation as seen with Method 2 and affect the threshold value that will be mentioned as part of Future Works. This answers the first part of the key question: “What is this kinematics structure”.

2. Expanding the existing joint angle calculation

Before this thesis was written, a joint angle calculation was introduced for one unit cell. This calculation was expanded to consider an array of joints. The calculation was successful because the size of the matrix would not affect how the calculation is implemented: the calculation can calculate any matrix size.

3. Incorporating the joint stiffness

From the material research and the experimental run, we were able to calculate the stiffness property and apply in our equation to determine the deformation. Although we were able to derive the stiffness values for both types of joints, of all of the goals, this is the least successful. The experimental set-up did not have the proper tools to measure angular springs that only needs up to 1N for the displacement response. From what we had, we were successful in setting-up and evaluating the stiffness value since the angle of deformation was not high.

4. Developing a “Forward Statics” Algorithm

From Method 2, we have completed all the goals by developing the forward Statics equations to complete the cycle with inverse Statics with material consideration. The algorithm solves the problem and partially answered the second part of the key question.

Second part of key question: “how can the deformation of the kinematics structure be predicted based upon the materials being used and constraints being applied?” As mentioned, there are two methods presented with 3 different implementations total. Both methods consider the material being used by using the stiffness value in the calculation. Both methods also consider the constraints being applied by the user to predict the deformation. It is up to the user which method and implementation to use. Each method can be extrapolated for other crust designs with angles as the deformation feature. Also both the methods and the implementations are robust. The implementations are robust because any of the implementation can handle various size matrix and unit cells with different constraints being applied. The methods are robust because the methods can also handle any matrix and unit cell size. As previously mentioned, the starting position can also vary and the methods will arrive at an answer. Of course, there is a question of uniqueness that was previously addressed.

Both parts of the keys questions are successfully answered while fulfilling the goals of this thesis.

6.2 Future Work

Below are suggested improvements that can be made to this thesis.

1. Implement Method 2 for Matrix

Currently the programs that implement Method 2 only model a serial chain of unit cells. The programs can be modified to create a whole matrix by considering the geometry as each unit cell gets connected to each other and more realization of duplication of the variables in the matrix. The question one has to ask is what variables are being duplicated and how are they duplicated.

2. Methods for Hex Matrix

Extrapolating the methods developed for the Grid Matrix to the Hex Matrix. The math for the Hex Matrix has not been developed yet because it was easier to start off with the Grid Matrix. After developing a system to analyze the Grid Matrix, it will now be easier to analyze the Hex Matrix.

3. Looping for Shape Editing

As previously explained, any of the methods developed can be used for Shape Editing. That means that one shape can transform into a different shape by modifying the inputs. Any of the programs created to implement the methods can be changed to support Shape Editing by adding a loop command within the program. This modification has not yet been completed due to time constraints.

4. Converting MATLAB to C++

MATLAB is a higher level programming language than C++; however C++ has higher computational speed. C++ also allows the programmer to create a better, friendlier user interface than MATLAB. At present, I, the Mechanical Engineer, am working with a Computer Science Master student, to convert the MATLAB code that implemented the methods into C++. Presently, this process has not been completed.

5. Exploration of Material

For this development thesis, a stereolithography material was used to create the matrices. In the MEMS department other materials are being explored. Presently, we have not explored enough material to determine which material is best fitted to build the formable crust matrix that will be part of the Digital Clay device.

6. Adding Threshold Values

As stated in the last chapter, there needs to be a threshold value for the crust matrix as it deforms. This threshold value can be the maximum angle(s) deformation for any angle in the methods. If the threshold value is reached, there can be a feedback statement in the programs stating that the crust matrix will break. To include this criterion, we can add a loop at the end of the program that will evaluate each angle against the threshold angle(s). The threshold angle(s) is/are determined by the design of each unit cell and can be experimentally estimated.

Another criterion is calculating the energy stored in each spring. If the energy stored in a spring is greater than a threshold value, then the constraints imposed by the method being used and user input values would cause the crust matrix to break. For example, a piece of wood cannot handle a Z-height input applied in the middle while being constrained at its corners without breaking. The threshold value varies with the material being used.

These new criteria are easy to attach at the end of the existing methods. However, the threshold values for these criteria require more research in material and experimental runs.

7. Experimentation Validation

The previous chapter shows various simulation runs of the implementations of the two different methods. However, the results are not validated. This means we do not have a machine that would confirm the cartesian coordinate values of every center-point of every unit cells in a matrix. A machine can be built to measure the X, Y, and Z coordinate values before and after deformation.

8. Exploration of design

Although the current design is very well done in creating various shapes, there can possibly be other designs that can deform better and be easier to manufacture.

6.3 Benefits and Values

For each subgroup, this thesis will provide some potential benefits. There are six subgroups. They are MEMS headed by Dr. Mark Allen, Controls headed by Dr. Wayne Book, Fluids by Dr. Ari Glezer, Manufacturing by Dr. David Rosen, Interface by Dr. Jarek Rossignac, and Kinematics by Dr. Imme Ebert Uphoff.

1. MEMS

Currently the MEMS group is investigating various materials to build the formable crust matrix valves. Using the methods developed for this thesis, the MEMS department can determine which material is best fitted for creating the desired deformation. Also, any of the programs developed can output the deformed angles for any unit cell in the matrix. The MEMS department can use the output to determine the maximum angle deformation for any given shape. Once the maximum angle deformation is found, the optimal valve can be designed for the crust matrix.

2. Controls

The Control group led by Dr. Wayne Book is investigating various devices to manipulate the crust matrix once it is developed. With the advent of the methods developed for this thesis, the control department can predict (or control) what shape will be outputted based upon the inputs.

3. Fluids

Since the Digital Clay device will have fluids enclosed within the hardware to expand and contract the formable crust matrix, the Fluids group can apply the methods developed to determine how much fluid is needed. The amount of fluid will affect the force applied to the formable crust matrix, which in turn affects the shape configuration.

4. Manufacturing

Currently we are investigating the various materials to build the formable crust. The materials include different resins for the SLA technique. Currently we decided on DSM 8120 because of its elastic modulus material property, which in turn affects the stiffness values of the joints. This does not mean we have ended our material search. The methods developed will help us select other potential materials.

5. Interface

Currently we are collaborating with the Computer Science group to convert the MATLAB code. The programs will help the C.S. department advance their development for creating a program that can externally manipulate the formable crust matrix and start developing the interface for the Digital Clay device.

6. Kinematics

The methods developed for predicting the shape output applies several statics principles. That includes inverse and forward statics and spherical coordinate matrix manipulation. These statics principles can advance the Kinematics group's current understanding of the formable crust matrix.

APPENDIX A:

FINDING STIFFNESS VALUE

Through various experimental runs, the joint stiffness values for the two types of joints that comprise one unit cell are found. This appendix will contain graphical results from these runs.

Graphs for Large Joint

Run 1

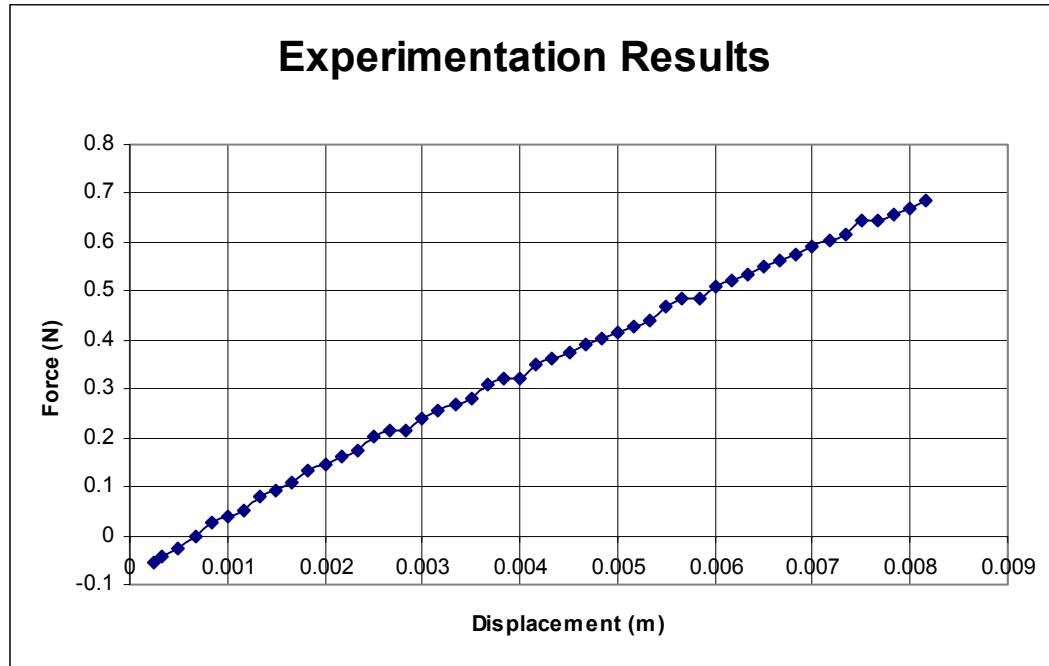


Figure A.1: Results Large Joint Run 1

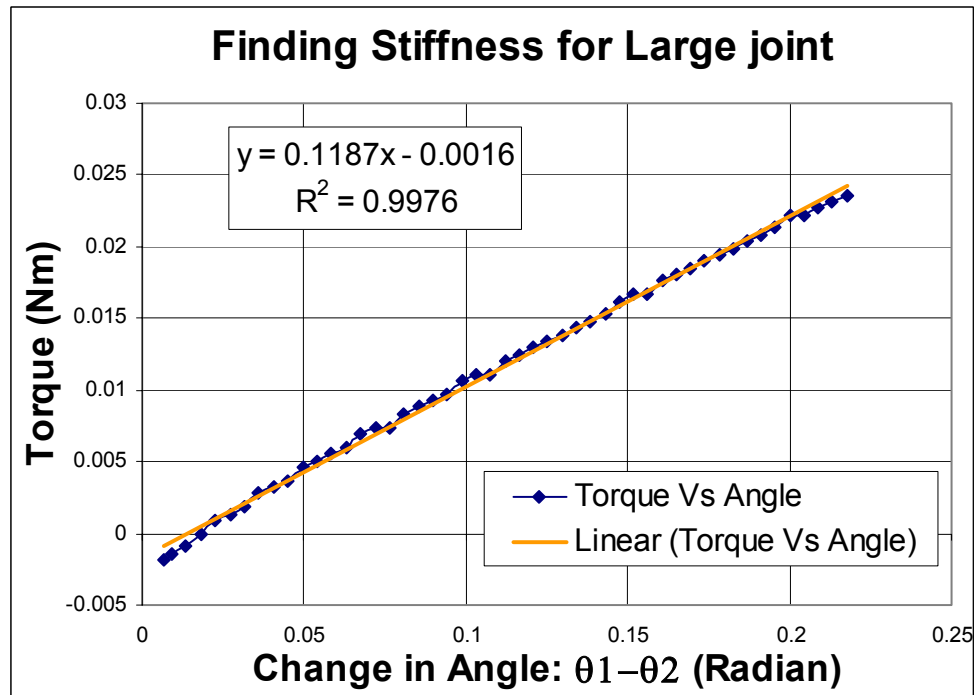


Figure A.2: Stiffness Value for Large Joint. Run 1

Run 2

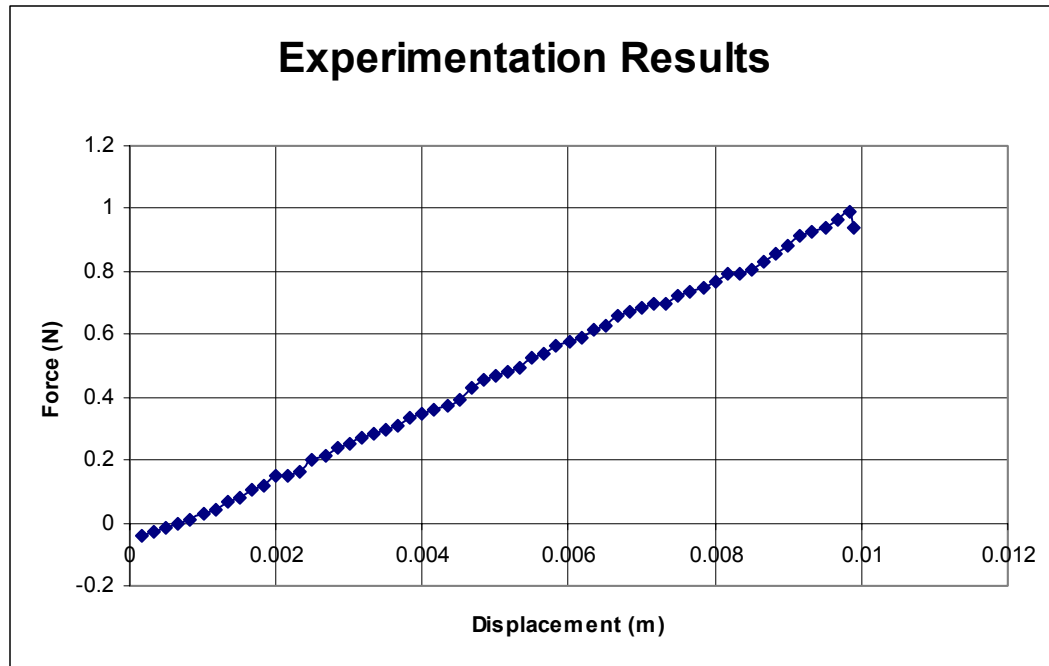


Figure A.3: Results Large Joint Run 2

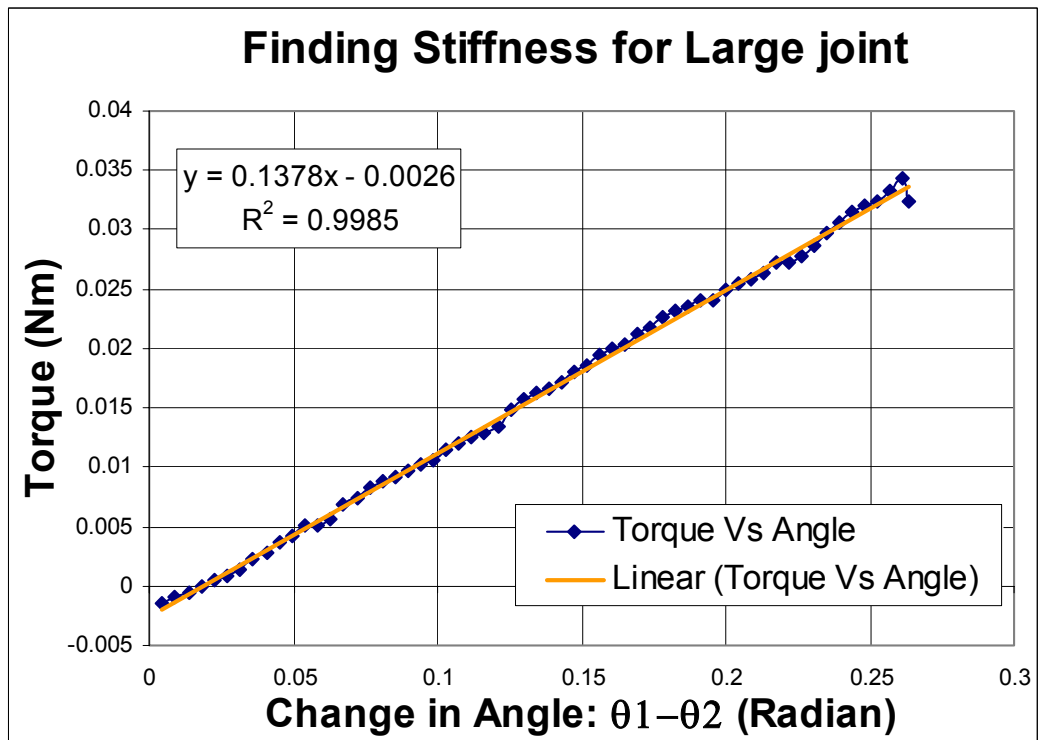


Figure A.4: Stiffness Value for Large Joint. Run 2

Run 3

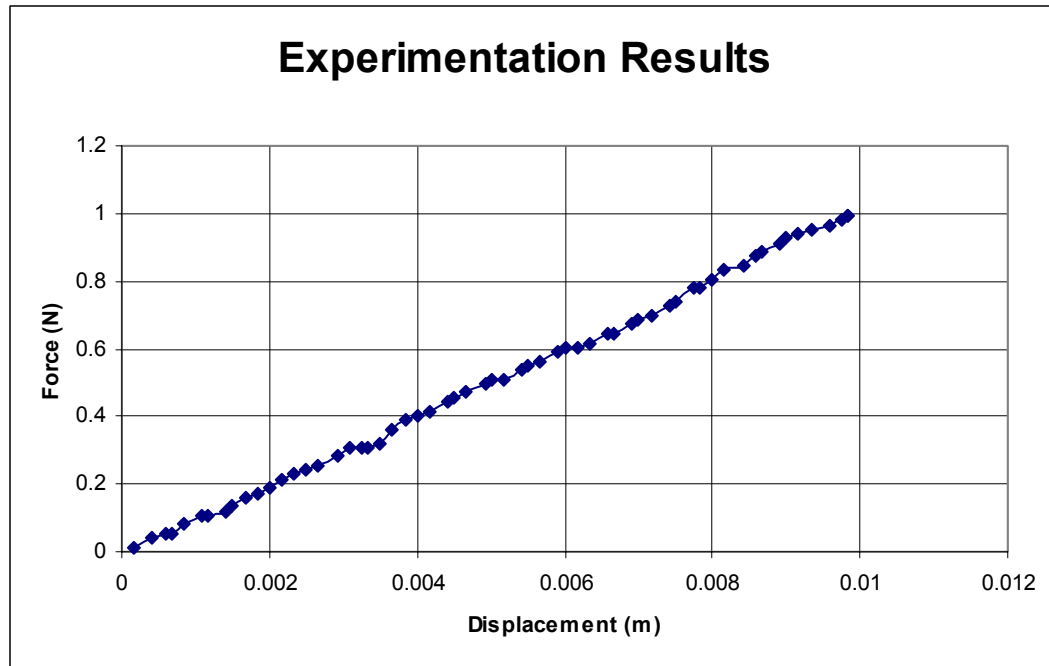


Figure A.5: Results Large Joint Run 3

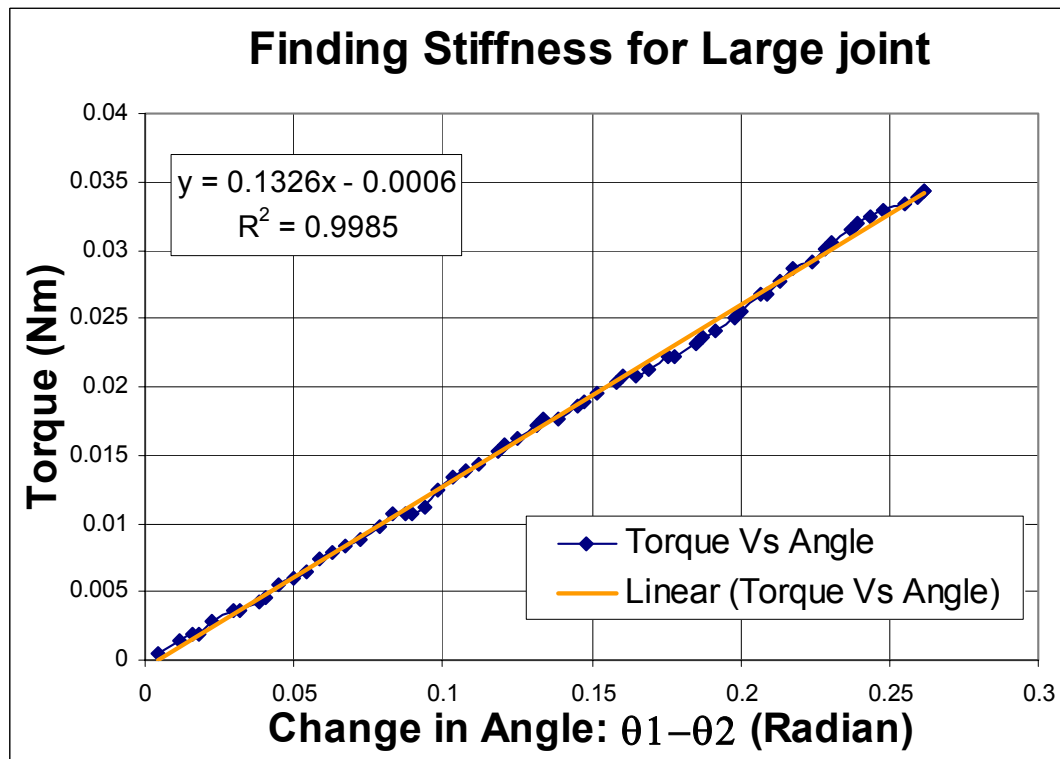


Figure A.6: Stiffness Value for Large Joint. Run 3

Run 4

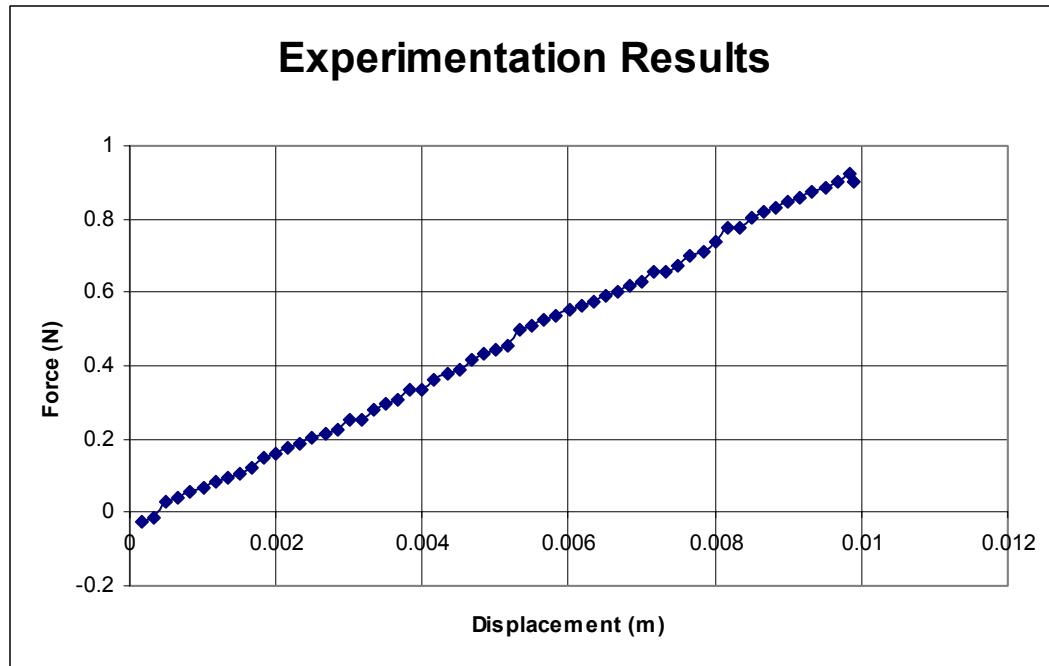


Figure A.7: Results Large Joint Run 4

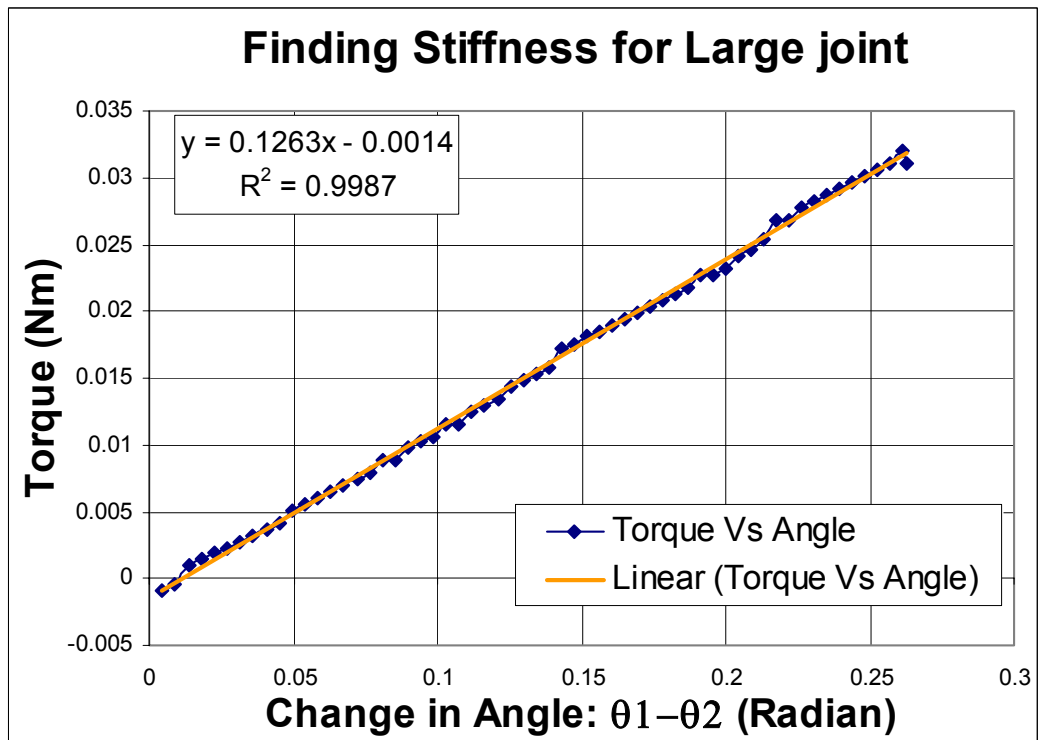


Figure A.8: Stiffness Value for Large Joint. Run 4

Run 5

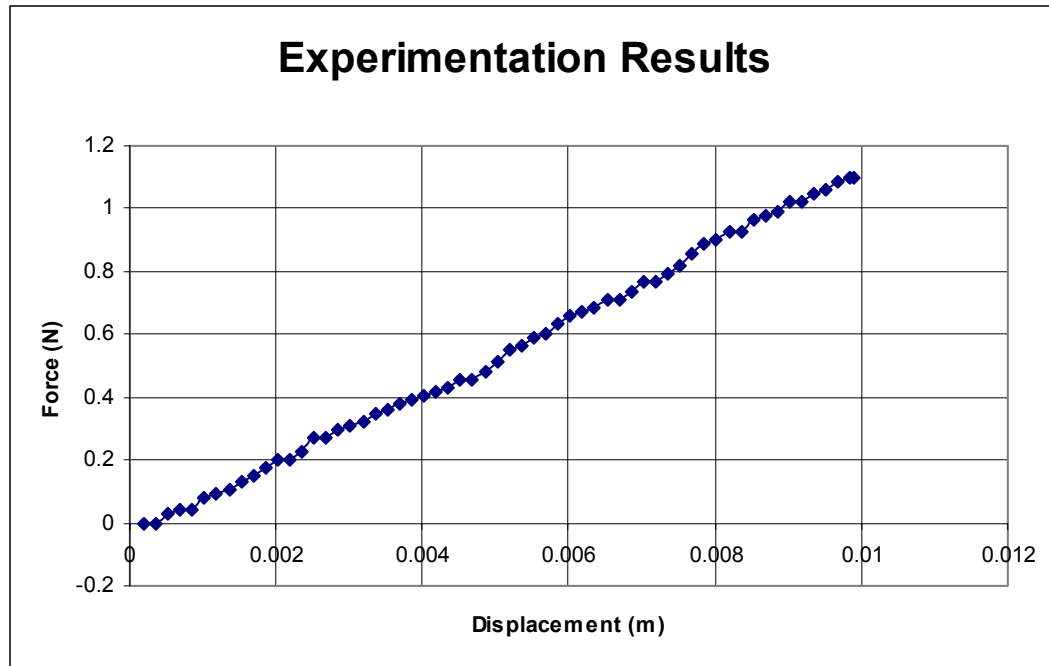


Figure A.9: Results Large Joint Run 5

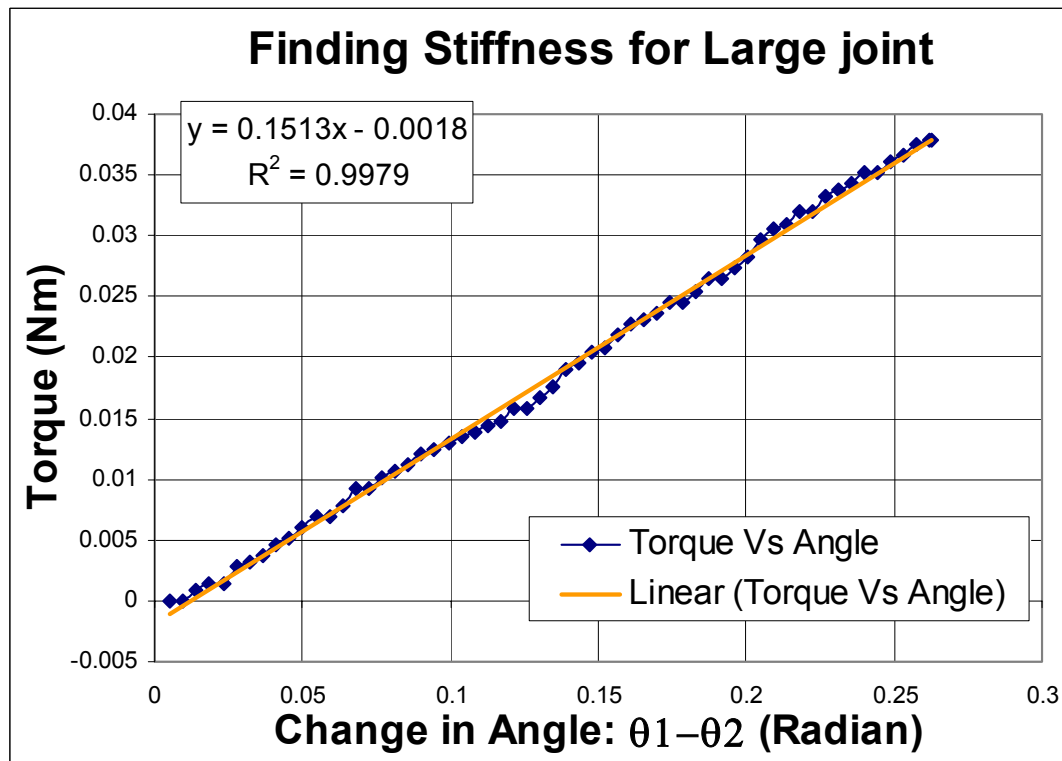


Figure A.10: Stiffness Value for Large Joint. Run 5

Run 6

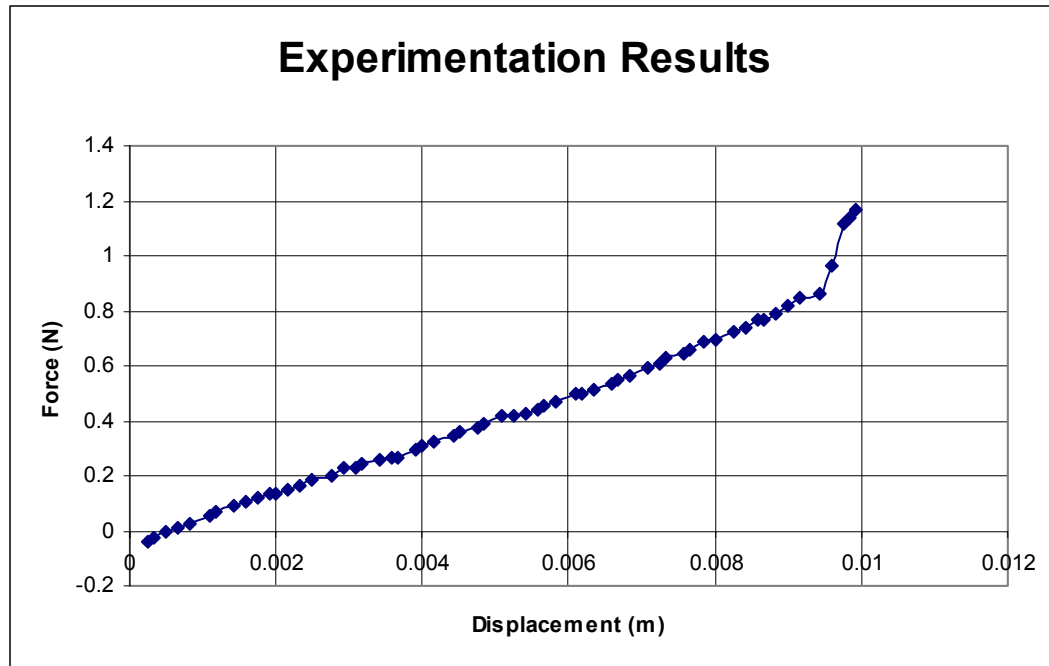


Figure A.11: Results Large Joint Run 6

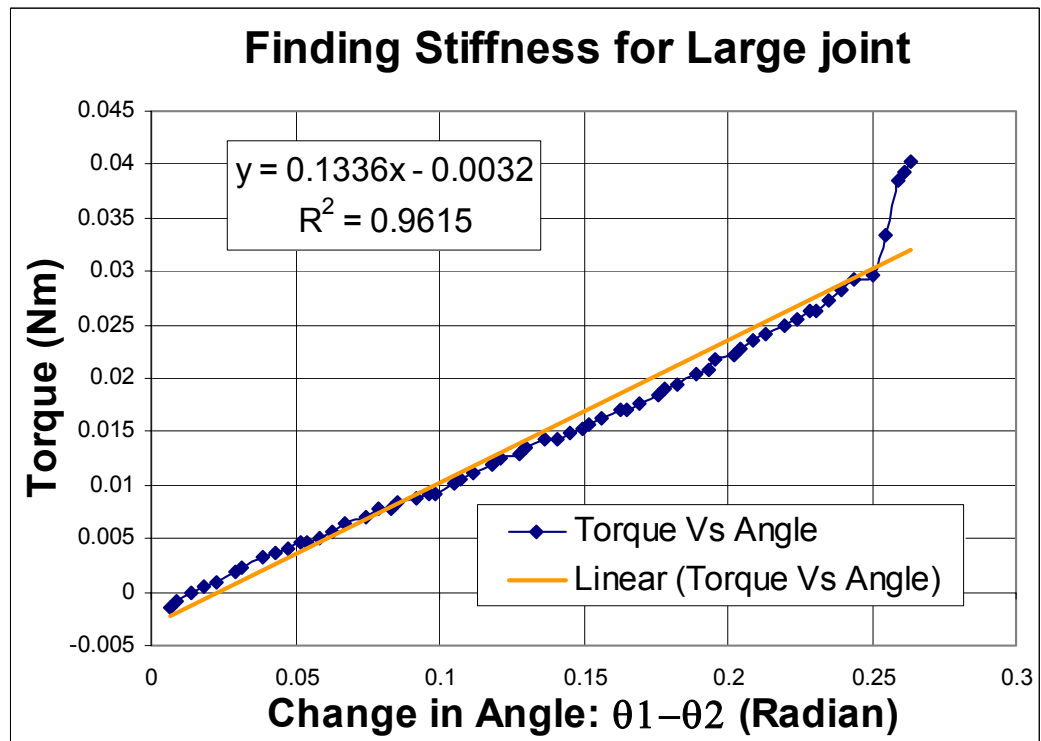


Figure A.12: Stiffness Value for Large Joint. Run 6

Run 7

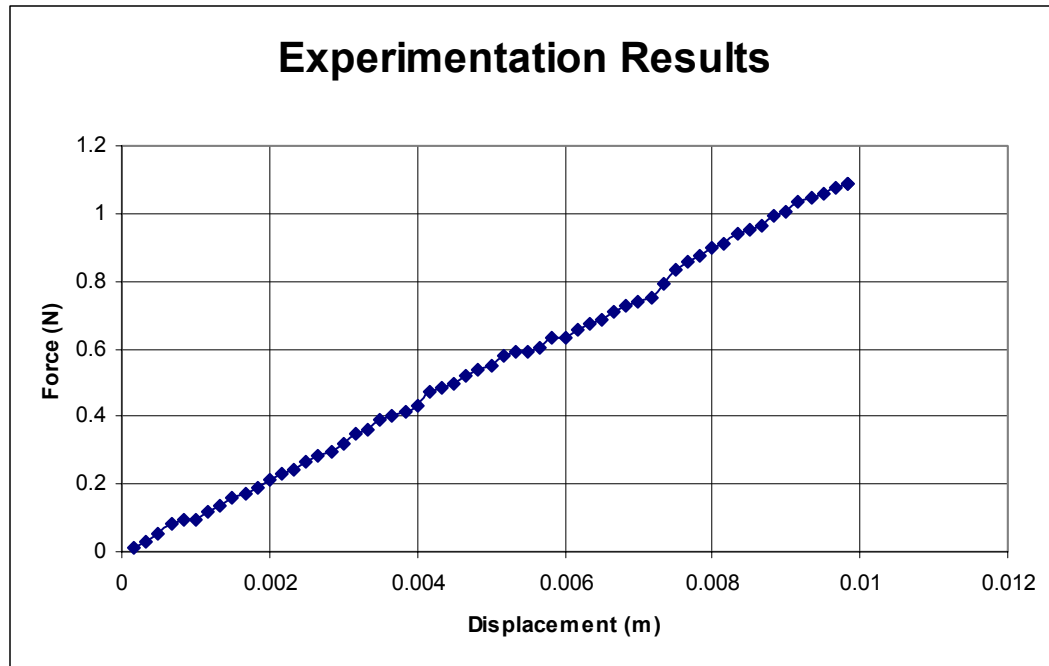


Figure A.13: Results Large Joint Run 7

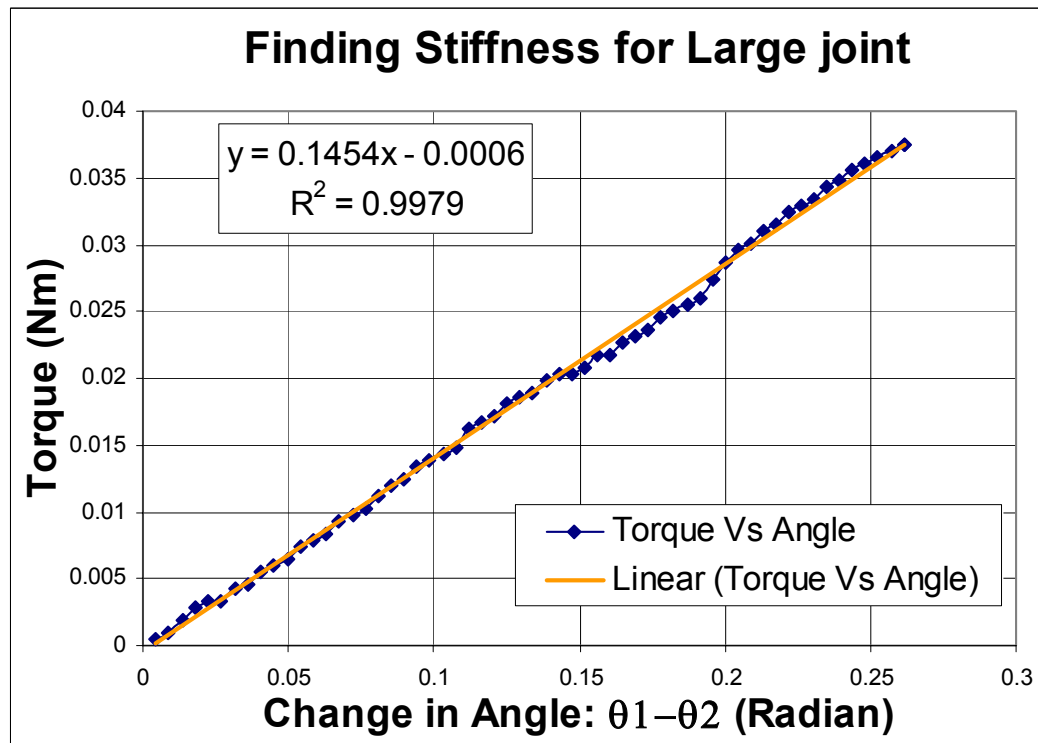


Figure A.14: Stiffness Value for Large Joint. Run 7

Run 8

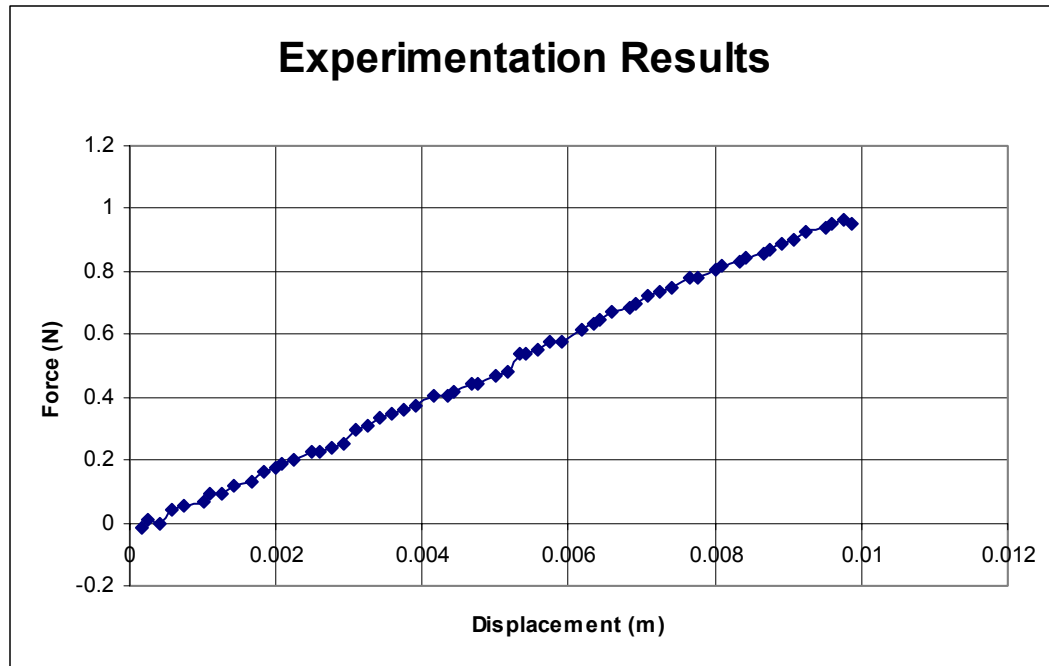


Figure A.15: Results Large Joint Run 8

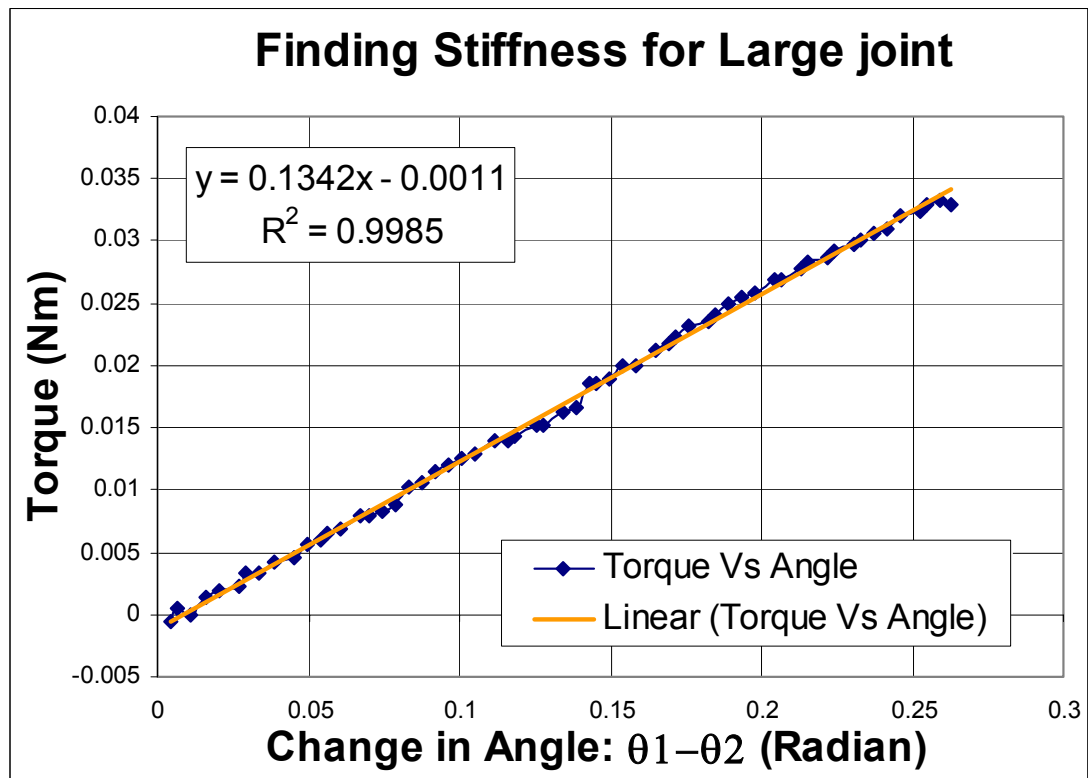


Figure A.16: Stiffness Value for Large Joint. Run 8

Run 9

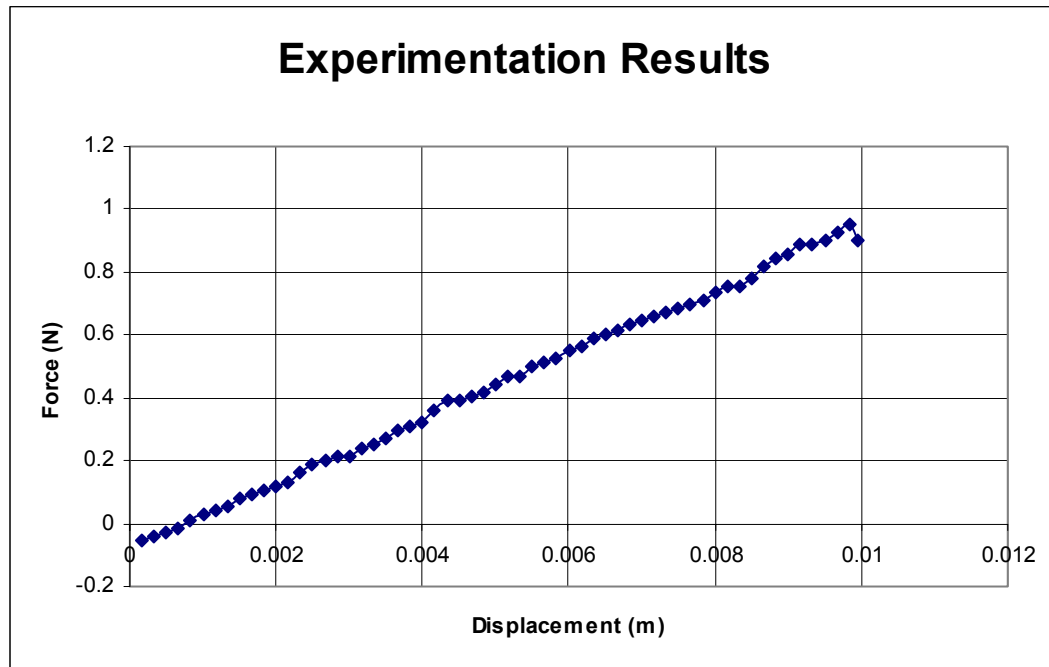


Figure A.17: Results Large Joint Run 9

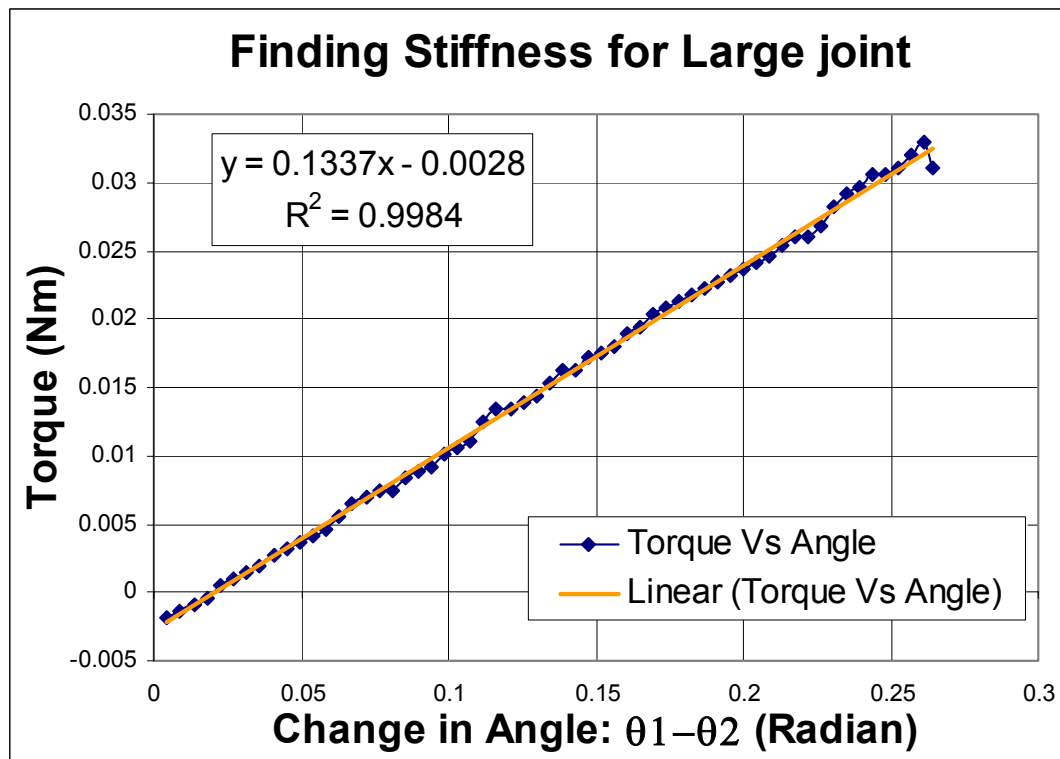


Figure A.18: Stiffness Value for Large Joint. Run 9

Table A.1: Summary of Stiffness Values for Large Joint

EXPERIMENTAL RUN FOR LARGE JOINT	STIFFNESS VALUE
RUN 1	0.1187
RUN 2	0.1378
RUN 3	0.1326
RUN 4	0.1263
RUN 5	0.1513
RUN 6	0.1336
RUN 7	0.1454
RUN 8	0.1342
RUN 9	0.1337
AVERAGE:	0.1348

SMALL JOINT

Run 1

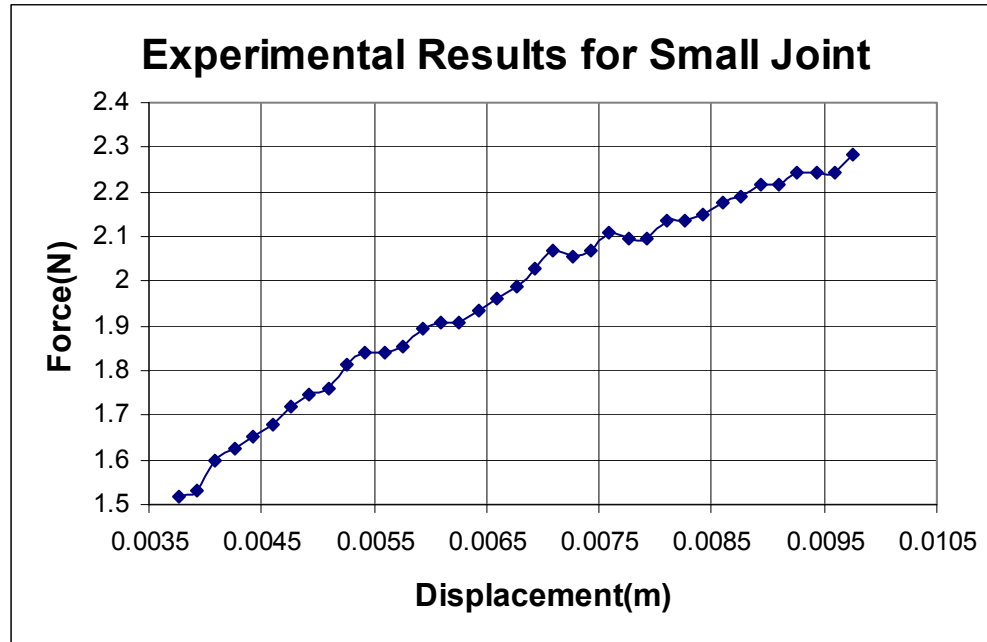


Figure A.19: Results Small Joint Run 1

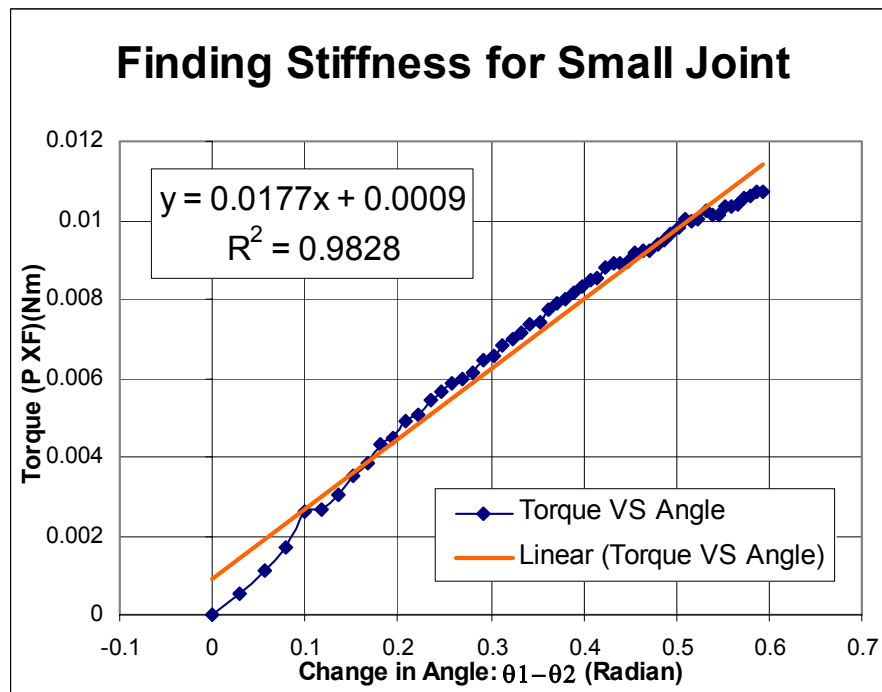


Figure A.20: Stiffness Value for Small Joint. Run 1

Run 2

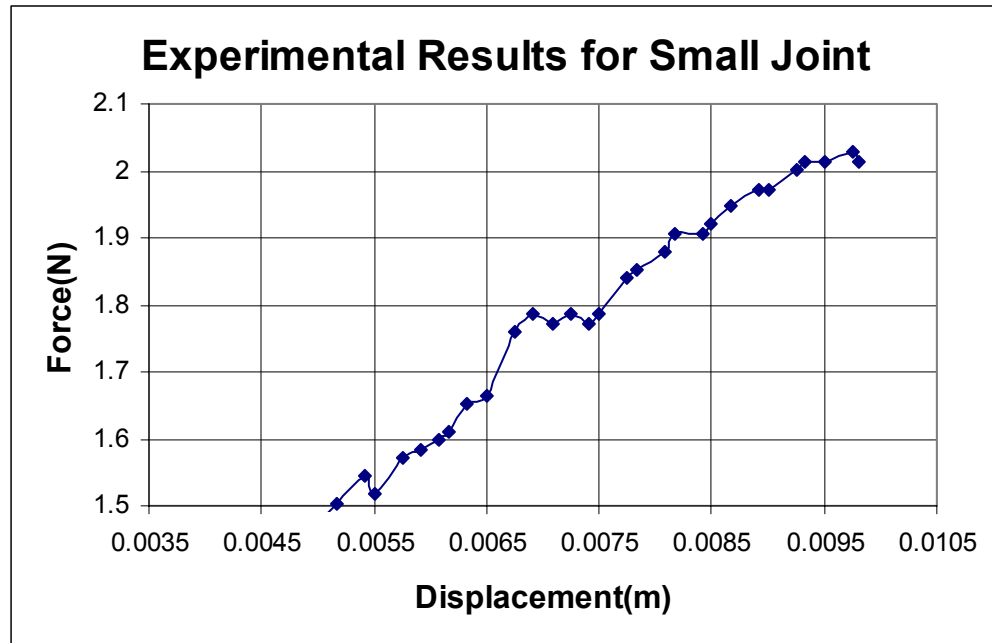


Figure A.21: Results Small Joint Run 2

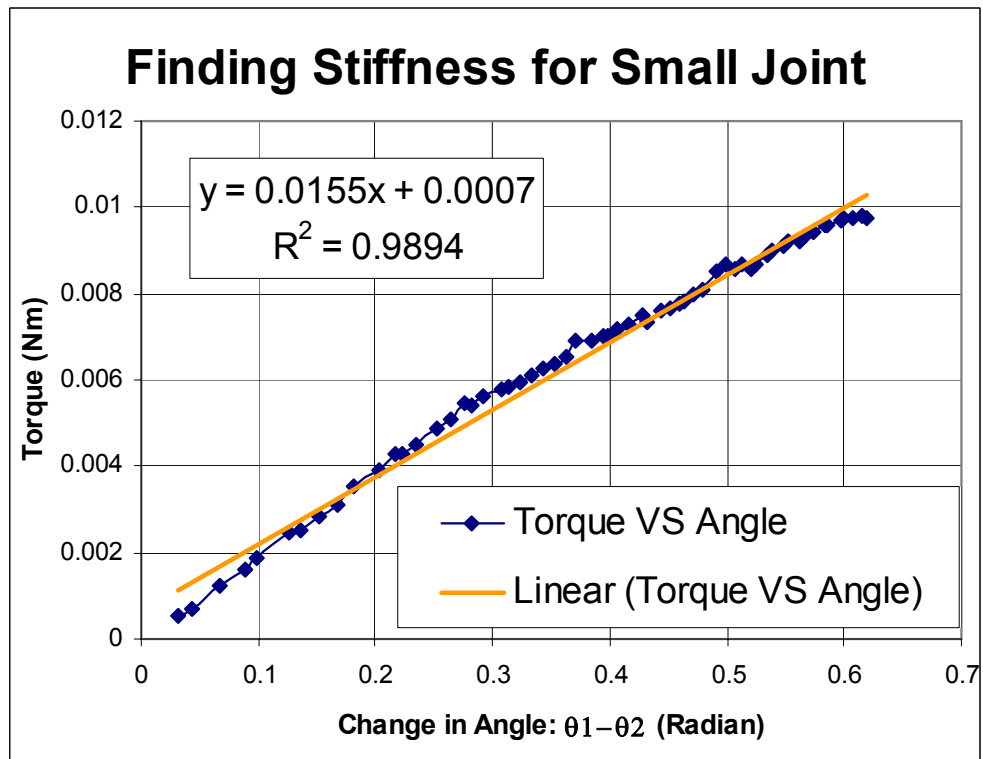


Figure A.22: Stiffness Value for Small Joint. Run 2

Run 3

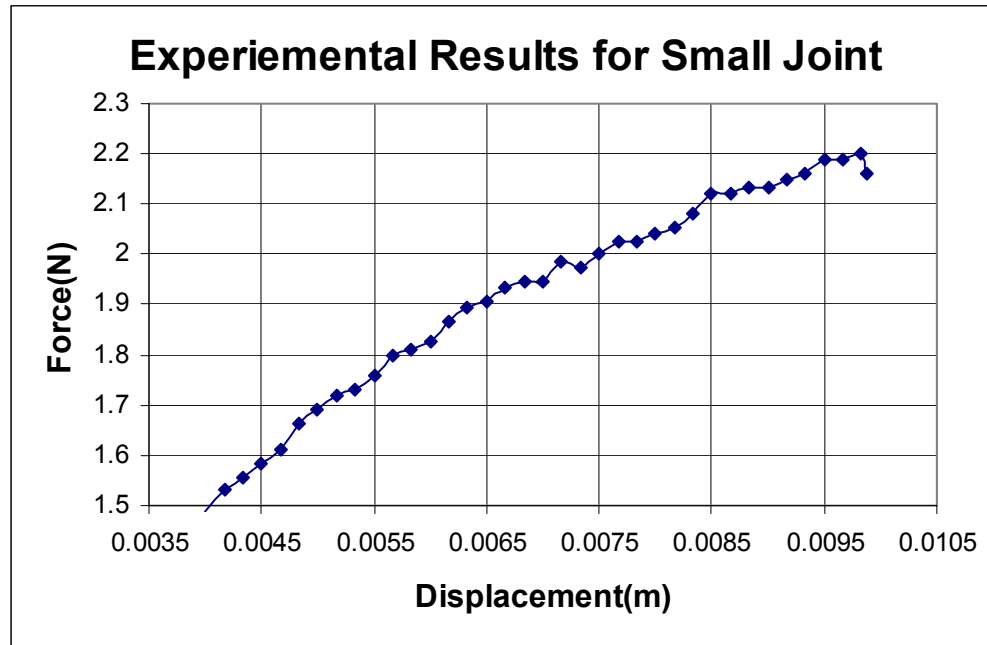


Figure A.23: Results Small Joint Run 3

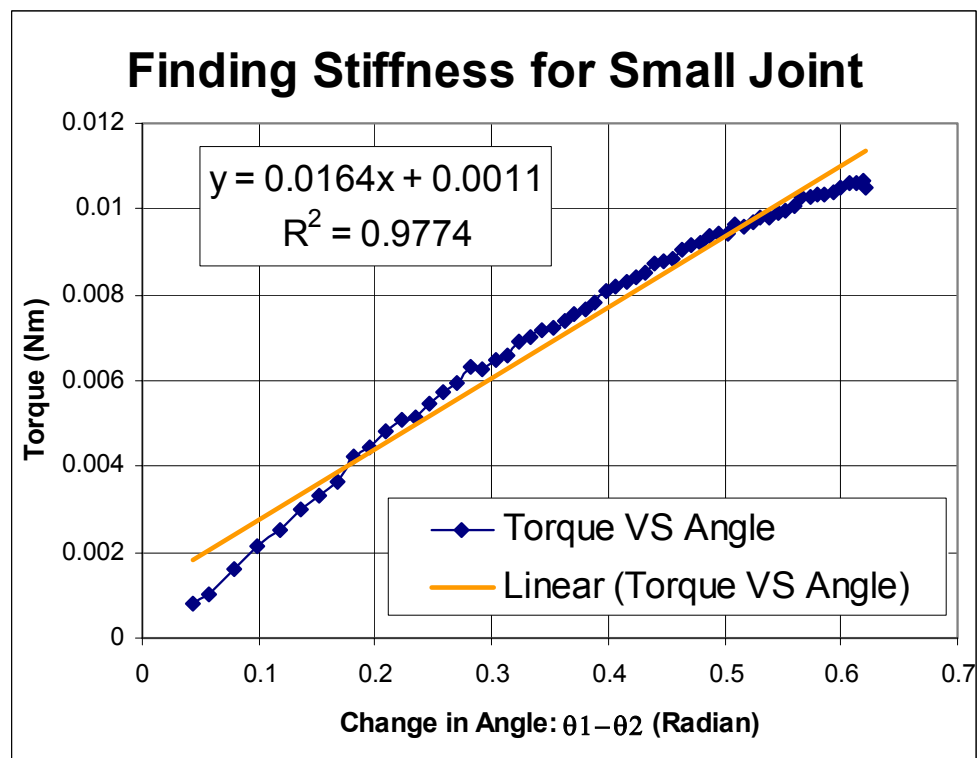


Figure A.24: Stiffness Value for Small Joint. Run 3

Run 4

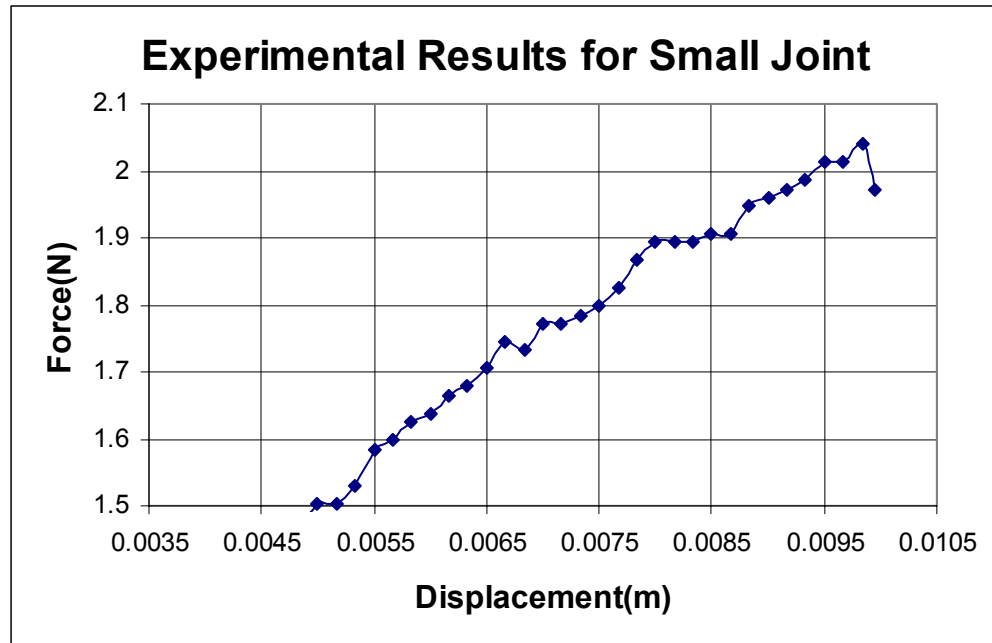


Figure A.25: Results Small Joint Run 4

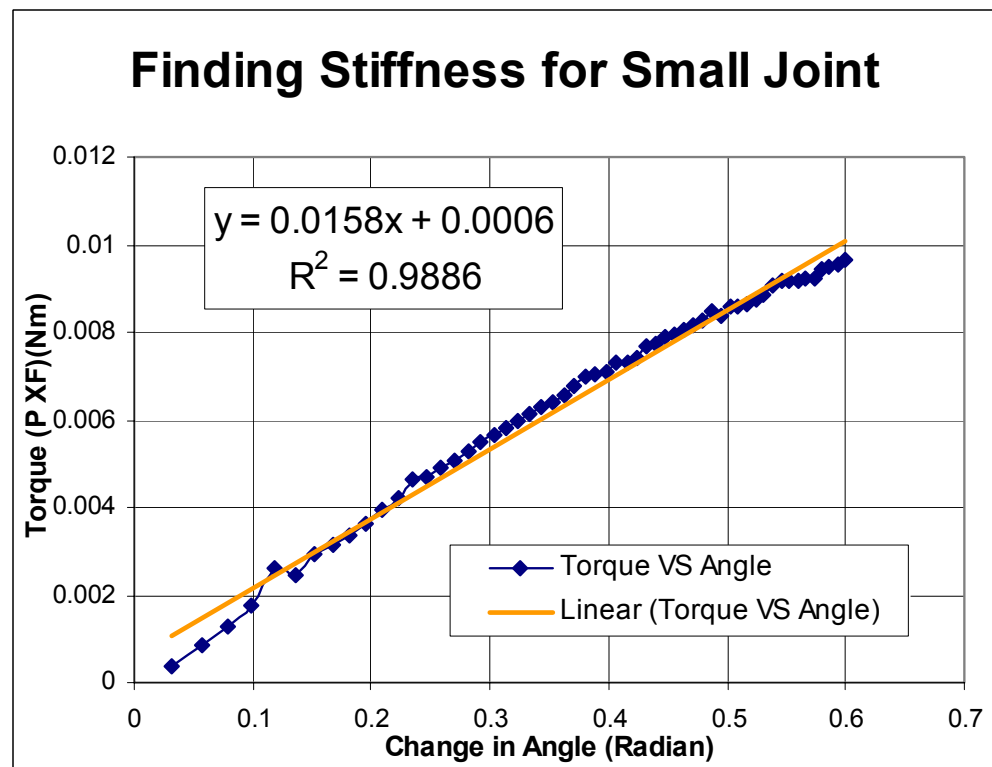


Figure A.26: Stiffness Value for Small Joint. Run 4

Table A.2: Summary of Stiffness Values for Small Joint

EXPERIMENTAL RUN FOR SMALL JOINT	STIFFNESS VALUE
RUN 1	0.0177
RUN 2	0.0155
RUN 3	0.0164
RUN 4	0.0158
AVERAGE:	0.01635

There were originally 6 runs, but only four are shown because the other 2 runs were outliers. These other runs will give false information and are deleted from this section.

APPENDIX B:

JOINT ANGLES DEFORMATION

There are three programs that implement the two different methods. From the results from each method, one can calculate the joint angles for every cell. For a 1-by-7 there are 7 cells and 12 joint angles for each cell. This appendix will show the joint angles before and after deformation and the energy stored in them.

First Program: Method 1

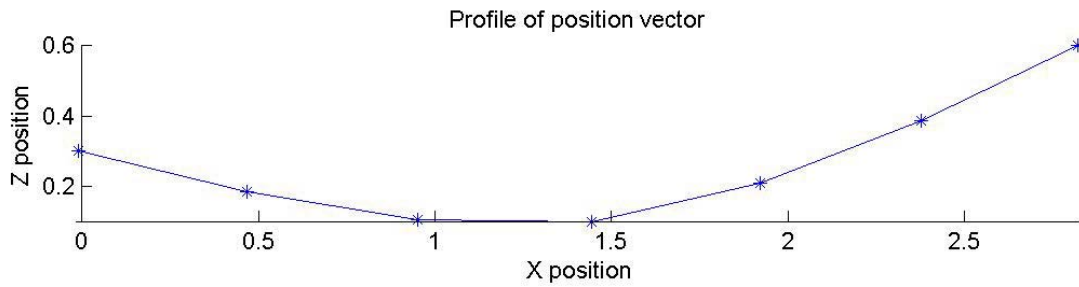


Figure B.1: Results from Method 1

Inputs: First cell: 0.3; Fourth cell:0.1; Last cell:0.6

Legend:

After: Joint angles (radian) after the program converges

Before: Joint angles (radian) before the program converges

Dif: the absolute value difference between the before and after angles

Energy: the energy stored in the spring: $k \cdot (\text{after} - \text{before})^2$

K: Stiffness Value: 0.8375Nm

The alphabet letters denotes the symmetry of the angles: For example: All A values for the Cell 1 has similar values in Method 1. All B in Cell 1 has similar values, etc.

Table B.1: Joint Results For Cell 1 and 2. Method 1

	Cell 1					Cell 2			
	After	Before	Dif	Energy		After	Before	Dif	Energy
A	1.7592	2.0089	0.2497	0.005222	A	1.8381	2.0089	0.1708	0.002443
B	1.4507	1.4432	0.0075	4.71E-06	B	1.4467	1.4432	0.0035	1.03E-06
C	2.0604	2.0089	0.0515	0.000222	C	2.0435	2.0089	0.0346	0.0001
C	2.0604	2.0089	0.0515	0.000222	C	2.0435	2.0089	0.0346	0.0001
B	1.4507	1.4432	0.0075	4.71E-06	B	1.4467	1.4432	0.0035	1.03E-06
A	1.7592	2.0089	0.2497	0.005222	A	1.8381	2.0089	0.1708	0.002443
D	2.0089	2.0089	0	0	D	2.2643	2.0089	0.2554	0.005463
E	1.4432	1.4432	0	0	E	1.4507	1.4432	0.0075	4.71E-06
F	2.0089	2.0089	0	0	F	1.9632	2.0089	0.0457	0.000175
F	2.0089	2.0089	0	0	F	1.9632	2.0089	0.0457	0.000175
E	1.4432	1.4432	0	0	E	1.4507	1.4432	0.0075	4.71E-06
D	2.0089	2.0089	0	0	D	2.2643	2.0089	0.2554	0.005463

Table B.2: Joint Results For Cell 3 and 4. Method 1

	Cell 3					Cell 4			
	After	Before	Dif	Energy		After	Before	Dif	Energy
A	1.9991	2.0089	0.0098	8.04E-06	A	2.2468	2.0089	0.2379	0.00474
B	1.4432	1.4432	0	0	B	1.4497	1.4432	0.0065	3.54E-06
C	2.0108	2.0089	0.0019	3.02E-07	C	1.9661	2.0089	0.0428	0.000153
C	2.0108	2.0089	0.0019	3.02E-07	C	1.9661	2.0089	0.0428	0.000153
B	1.4432	1.4432	0	0	B	1.4497	1.4432	0.0065	3.54E-06
A	1.9991	2.0089	0.0098	8.04E-06	A	2.2468	2.0089	0.2379	0.00474
D	2.1824	2.0089	0.1735	0.002521	D	2.0187	2.0089	0.0098	8.04E-06
E	1.4467	1.4432	0.0035	1.03E-06	E	1.4432	1.4432	0	0
F	1.9769	2.0089	0.032	8.58E-05	F	2.007	2.0089	0.0019	3.02E-07
F	1.9769	2.0089	0.032	8.58E-05	F	2.007	2.0089	0.0019	3.02E-07
E	1.4467	1.4432	0.0035	1.03E-06	E	1.4432	1.4432	0	0
D	2.1824	2.0089	0.1735	0.002521	D	2.0187	2.0089	0.0098	8.04E-06

Table B.3: Joint Results For Cell 5 and 6. Method 1

	Cell 5					Cell 6			
	After	Before	Dif	Energy		After	Before	Dif	Energy
A	2.4107	2.0089	0.4018	0.013521	A	2.4917	2.0089	0.4828	0.019522
B	1.4614	1.4432	0.0182	2.77E-05	B	1.4692	1.4432	0.026	5.66E-05
C	1.9411	2.0089	0.0678	0.000385	C	1.9304	2.0089	0.0785	0.000516
C	1.9411	2.0089	0.0678	0.000385	C	1.9304	2.0089	0.0785	0.000516
B	1.4614	1.4432	0.0182	2.77E-05	B	1.4692	1.4432	0.026	5.66E-05
A	2.4107	2.0089	0.4018	0.013521	A	2.4917	2.0089	0.4828	0.019522
D	1.776	2.0089	0.2329	0.004543	D	1.6213	2.0089	0.3876	0.012582
E	1.4497	1.4432	0.0065	3.54E-06	E	1.4614	1.4432	0.0182	2.77E-05
F	2.0568	2.0089	0.0479	0.000192	F	2.0908	2.0089	0.0819	0.000562
F	2.0568	2.0089	0.0479	0.000192	F	2.0908	2.0089	0.0819	0.000562
E	1.4497	1.4432	0.0065	3.54E-06	E	1.4614	1.4432	0.0182	2.77E-05
D	1.776	2.0089	0.2329	0.004543	D	1.6213	2.0089	0.3876	0.012582

Table B.4: Joint Results For Cell 7. Method 1

	Cell 7			
	After	Before	Dif	Energy
A	2.0089	2.0089	0	0
B	1.4432	1.4432	0	0
C	2.0089	2.0089	0	0
C	2.0089	2.0089	0	0
B	1.4432	1.4432	0	0
A	2.0089	2.0089	0	0
D	1.5463	2.0089	0.4626	0.017922
E	1.4692	1.4432	0.026	5.66E-05
F	2.1076	2.0089	0.0987	0.000816
F	2.1076	2.0089	0.0987	0.000816
E	1.4692	1.4432	0.026	5.66E-05
D	1.5463	2.0089	0.4626	0.017922

Second Program: Method 2 Fix Face

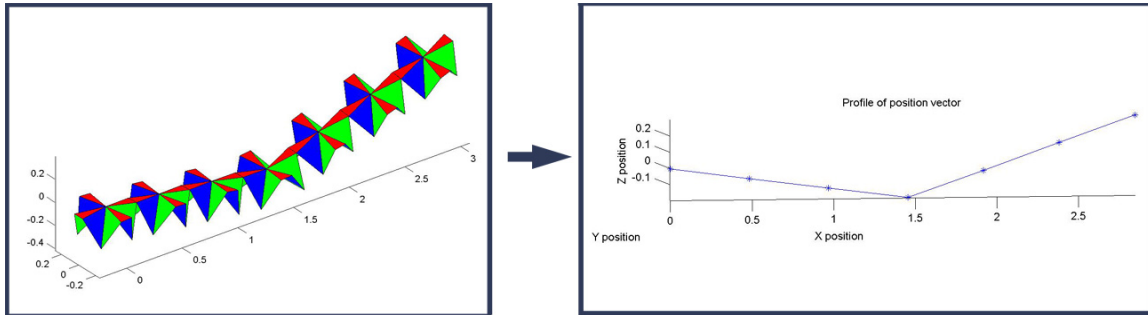


Figure B.2: Results from Method 2: Fix Face

Same inputs, same legend definition.

Stiffness: $k_1 = 0.152\text{Nm}$ and $k_2 = 0.0155\text{Nm}$

Energy Equation for larger joint: $0.152 * (\text{Before} - \text{After})^2$

Energy Equation for larger joint: $0.0155 * (\text{Before} - \text{After})^2$

Table B.5: Joint Results For Cell 1 and 2. Method 2: Fix

	Cell 1					Cell 2			
	After	Before	Dif	Energy		After	Before	Dif	Energy
	1.8021	2.0089	0.2068	0.0065	A	1.9541	2.0089	0.0548	0.000456
	1.4728	1.4432	0.0296	1.36E-05	B	1.4534	1.4432	0.0102	1.61E-06
	2.0049	2.0089	0.004	2.43E-06	C	2.0165	2.0089	0.0076	8.78E-06
	2.0171	2.0089	0.0082	1.04E-06	C	2.0125	2.0089	0.0036	2.01E-07
	1.4271	1.4432	0.0161	3.94E-05	B	1.4368	1.4432	0.0064	6.23E-06
	1.9407	2.0089	0.0682	7.21E-05	A	2.0069	2.0089	0.002	6.2E-08
	1.9323	2.0089	0.0766	0.000892	D	2.006	2.0089	0.0029	1.28E-06
	1.4367	1.4432	0.0065	6.55E-07	E	1.4393	1.4432	0.0039	2.36E-07
	1.9801	2.0089	0.0288	0.000126	F	2.003	2.0089	0.0059	5.29E-06
	1.9749	2.0089	0.034	1.79E-05	F	2.0007	2.0089	0.0082	1.04E-06
	1.4351	1.4432	0.0081	9.97E-06	E	1.4442	1.4432	0.001	1.52E-07
	1.9471	2.0089	0.0618	5.92E-05	D	2.0059	2.0089	0.003	1.4E-07

Table B.6: Joint Results For Cell 3 and 4. Method 2: Fix

Cell 3					Cell 4				
	After	Before	Dif	Energy		After	Before	Dif	Energy
A	2.0231	2.0089	0.0142	3.06E-05	A	2.2246	2.0089	0.2157	0.007072
B	1.4437	1.4432	0.0005	3.87E-09	B	1.4079	1.4432	0.0353	1.93E-05
C	2.0584	2.0089	0.0495	0.000372	C	2.1227	2.0089	0.1138	0.001968
C	2.008	2.0089	0.0009	1.26E-08	C	2.0378	2.0089	0.0289	1.29E-05
B	1.4636	1.4432	0.0204	6.33E-05	B	1.4295	1.4432	0.0137	2.85E-05
A	2.0211	2.0089	0.0122	2.31E-06	A	2.2151	2.0089	0.2062	0.000659
D	2.0228	2.0089	0.0139	2.94E-05	D	2.2272	2.0089	0.2183	0.007244
E	1.4621	1.4432	0.0189	5.54E-06	E	1.4068	1.4432	0.0364	2.05E-05
F	2.0134	2.0089	0.0045	3.08E-06	F	2.0997	2.0089	0.0908	0.001253
F	2.0448	2.0089	0.0359	2E-05	F	2.0514	2.0089	0.0425	2.8E-05
E	1.4039	1.4432	0.0393	0.000235	E	1.4126	1.4432	0.0306	0.000142
D	2.2058	2.0089	0.1969	0.000601	D	2.2104	2.0089	0.2015	0.000629

Table B.7: Joint Results For Cell 5 and 6. Method 2: Fix

Cell 5					Cell 6				
	After	Before	Dif	Energy		After	Before	Dif	Energy
A	2.1875	2.0089	0.1786	0.004848	A	2.0136	2.0089	0.0047	3.36E-06
B	1.4219	1.4432	0.0213	7.03E-06	B	1.4411	1.4432	0.0021	6.84E-08
C	2.0085	2.0089	0.0004	2.43E-08	C	2.0139	2.0089	0.005	3.8E-06
C	1.9607	2.0089	0.0482	3.6E-05	C	2.0026	2.0089	0.0063	6.15E-07
B	1.4639	1.4432	0.0207	6.51E-05	B	1.4455	1.4432	0.0023	8.04E-07
A	2.0082	2.0089	0.0007	7.6E-09	A	2.006	2.0089	0.0029	1.3E-07
D	2.0282	2.0089	0.0193	5.66E-05	D	2.0079	2.0089	0.001	1.52E-07
E	1.4345	1.4432	0.0087	1.17E-06	E	1.4422	1.4432	0.001	1.55E-08
F	2.0606	2.0089	0.0517	0.000406	F	2.0142	2.0089	0.0053	4.27E-06
F	1.9746	2.0089	0.0343	1.82E-05	F	2.0046	2.0089	0.0043	2.87E-07
E	1.4619	1.4432	0.0187	5.32E-05	E	1.4441	1.4432	0.0009	1.23E-07
D	2.0108	2.0089	0.0019	5.6E-08	D	2.0107	2.0089	0.0018	5.02E-08

Table B.8: Joint Results For Cell 7. Method 2: Fix

Cell 7				
	After	Before	Dif	Energy
A	2.0042	2.0089	0.0047	3.36E-06
B	1.4369	1.4432	0.0063	6.15E-07
C	1.9953	2.0089	0.0136	2.81E-05
C	1.9941	2.0089	0.0148	3.4E-06
B	1.4367	1.4432	0.0065	6.42E-06
A	2.0075	2.0089	0.0014	3.04E-08
D	2.013	2.0089	0.0041	2.56E-06
E	1.4281	1.4432	0.0151	3.53E-06
F	2.0249	2.0089	0.016	3.89E-05
F	1.968	2.0089	0.0409	2.59E-05
E	1.4722	1.4432	0.029	0.000128
D	1.7922	2.0089	0.2167	0.000728

Third Program: Method 2 Free Face

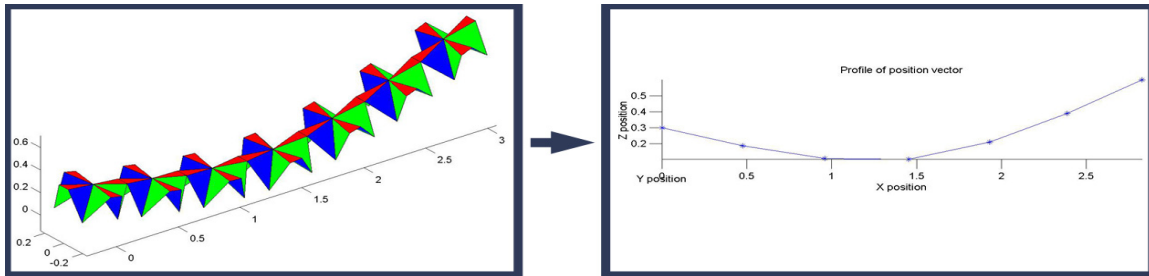


Figure B.3: Results from Method 2 Free Face

Same inputs, same legend definition.

Stiffness: $k_1 = 0.152\text{Nm}$ and $k_2 = 0.0155\text{Nm}$

Energy Equation for larger joint: $0.152 \cdot (\text{Before} - \text{After})^2$

Energy Equation for larger joint: $0.0155 \cdot (\text{Before} - \text{After})^2$

Table B.9: Joint Results For Cell 1 and 2. Method 2: Free

	Cell 1					Cell 2			
	After	Before	Dif	Energy		After	Before	Dif	Energy
A	2.0089	2.0089	0	0	A	2.0445	2.0089	0.0356	0.000193
B	1.4434	1.4432	0.0002	6.2E-10	B	1.443	1.4432	0.0002	6.2E-10
C	2.0085	2.0089	0.0004	2.43E-08	C	2.016	2.0089	0.0071	7.66E-06
C	2.0092	2.0089	0.0003	1.39E-09	C	2.0169	2.0089	0.008	9.92E-07
B	1.443	1.4432	0.0002	6.08E-09	B	1.4424	1.4432	0.0008	9.73E-08
A	2.0088	2.0089	0.0001	1.55E-10	A	2.0448	2.0089	0.0359	2E-05
D	2.0088	2.0089	0.0001	1.52E-09	D	2.0445	2.0089	0.0356	0.000193
E	1.4433	1.4432	1E-04	1.55E-10	E	1.443	1.4432	0.0002	6.2E-10
F	2.0087	2.0089	0.0002	6.08E-09	F	2.0159	2.0089	0.007	7.45E-06
F	2.009	2.0089	1E-04	1.55E-10	F	2.017	2.0089	0.0081	1.02E-06
E	1.4431	1.4432	1E-04	1.52E-09	E	1.4424	1.4432	0.0008	9.73E-08
D	2.0089	2.0089	0	0	D	2.0447	2.0089	0.0358	1.99E-05

Table B.10: Joint Results For Cell 3 and 4. Method 2: Free

Cell 3					Cell 4				
	After	Before	Dif	Energy		After	Before	Dif	Energy
A	2.0814	2.0089	0.0725	0.000799	A	2.1199	2.0089	0.111	0.001873
B	1.4408	1.4432	0.0024	8.93E-08	B	1.4369	1.4432	0.0063	6.15E-07
C	2.0262	2.0089	0.0173	4.55E-05	C	2.0377	2.0089	0.0288	0.000126
C	2.0259	2.0089	0.017	4.48E-06	C	2.0373	2.0089	0.0284	1.25E-05
B	1.441	1.4432	0.0022	7.36E-07	B	1.4373	1.4432	0.0059	5.29E-06
A	2.0818	2.0089	0.0729	8.24E-05	A	2.1202	2.0089	0.1113	0.000192
D	2.0816	2.0089	0.0727	0.000803	D	2.1202	2.0089	0.1113	0.001883
E	1.441	1.4432	0.0022	7.5E-08	E	1.4373	1.4432	0.0059	5.4E-07
F	2.0255	2.0089	0.0166	4.19E-05	F	2.0372	2.0089	0.0283	0.000122
F	2.0266	2.0089	0.0177	4.86E-06	F	2.0378	2.0089	0.0289	1.29E-05
E	1.4406	1.4432	0.0026	1.03E-06	E	1.4368	1.4432	0.0064	6.23E-06
D	2.0814	2.0089	0.0725	8.15E-05	D	2.1198	2.0089	0.1109	0.000191

Table B.11: Joint Results For Cell 5 and 6. Method 2: Free

Cell 5					Cell 6				
	After	Before	Dif	Energy		After	Before	Dif	Energy
A	2.0805	2.0089	0.0716	0.000779	A	2.0437	2.0089	0.0348	0.000184
B	1.4408	1.4432	0.0024	8.93E-08	B	1.4426	1.4432	0.0006	5.58E-09
C	2.0257	2.0089	0.0168	4.29E-05	C	2.0164	2.0089	0.0075	8.55E-06
C	2.0257	2.0089	0.0168	4.37E-06	C	2.0164	2.0089	0.0075	8.72E-07
B	1.441	1.4432	0.0022	7.36E-07	B	1.4428	1.4432	0.0004	2.43E-08
A	2.0807	2.0089	0.0718	7.99E-05	A	2.0438	2.0089	0.0349	1.89E-05
D	2.0806	2.0089	0.0717	0.000781	D	2.0438	2.0089	0.0349	0.000185
E	1.441	1.4432	0.0022	7.5E-08	E	1.4428	1.4432	0.0004	2.48E-09
F	2.0257	2.0089	0.0168	4.29E-05	F	2.0164	2.0089	0.0075	8.55E-06
F	2.0258	2.0089	0.0169	4.43E-06	F	2.0164	2.0089	0.0075	8.72E-07
E	1.4408	1.4432	0.0024	8.76E-07	E	1.4427	1.4432	0.0005	3.8E-08
D	2.0805	2.0089	0.0716	7.95E-05	D	2.0438	2.0089	0.0349	1.89E-05

Table B.12: Joint Results For Cell 7. Method 2: Free

Cell 7				
	After	Before	Dif	Energy
A	2.0089	2.0089	0	0
B	1.4431	1.4432	1E-04	1.55E-10
C	2.0088	2.0089	0.0001	1.52E-09
C	2.0089	2.0089	0	0
B	1.4433	1.4432	1E-04	1.52E-09
A	2.0089	2.0089	0	0
D	2.0089	2.0089	0	0
E	1.4433	1.4432	1E-04	1.55E-10
F	2.0089	2.0089	0	0
F	2.0088	2.0089	0.0001	1.55E-10
E	1.4432	1.4432	0	0
D	2.0089	2.0089	0	0

APPENDIX C:

MATLAB FOR METHOD 1

Method 1 was implemented through using MATLAB Coding. The first page will be a guide to the code and the functions in the code. The rest of the pages in this Appendix are the code.

NAVIGATION OUTLINE

FIRST PROGRAM IMPLEMENTING METHOD 1

Nomenclature:

xcell: a matrix of the Cartesian coordinates of the center-points of the unit cells
ncells: the matrix dimension
xcellvar: controlling the constraints. Dimension size is the same as xcell.
dist: fixed distance value between any two point
k_joint: stiffness value
xcell_initial: initial value of xcell before the modifying any values.
Zinputs: user dislocatemen inputs for the z-values in xcell
k_alpha: geometric constant from design model
k_beta: geometric constant from design model
k_lambda: geometric constant from design model

STICKFIGURE

1. **User Inputs:** requesting user inputs to create an m-by-n matrix
2. xcell_initial = xcell
3. Creating the 'on/' 'off' matrix for controlling the constraints
 - a. Initiating an emptied matrix: xcellvar
 - i. Note: xcellvar will be a matrix of the same size as xcells
 - b. For any Cartesian coordinate variables fixed in xcell,
 - i. The corresponding value in Xcellvar=0
 - c. For any Cartesian variables free to change in xcell,
 - i. The corresponding value in Xcellvar Xcellvar=1
4. **Link_length:** calculating the distance between any two center-point
5. **Graph_ctpt:** graphing the center-points of the unit cells from xcell
6. **guess_interpolating:** linear interpolation function
7. Creating Initial Guess vector: x0
 - a. If xcellvar=1, place xcell value into x0
8. **Fmincon** –blackbox MATLAB function
 - a. **MinimizingEnergy:** calculating the potential energy in the system for Fmincon

- b. **Controlling_length:** controls the distance between two center-points based upon the user input :dist”
- 9. **xcell:** the final results of Cartesian coordinates
- 10. **Link_length:** calculating the distance between any two center-point
- 11. **Graph_ctpt:** graphing the center-points of the unit cells from xcell
- 12. $k_alpha = (58.054 * \pi) / 180$; % convert degree to radian
- 13. $k_beta = (22 * \pi) / 180$;
- 14. $k_lamda = (62.806 * \pi) / 180$;
- 15. **boundary_vectors:** creating extra unit cells surrounding the developed matrix for calculating V-vectors.
- 16. **cal_v_n:** calculating the v-vectors and the n-vectors for each cells
- 17. **calc_theta:** calculating the angles of each joint for each cell in the matrix
- 18. **bdpt_remove:** removing the extra unit cells

USER_INPUTS

1. Specify user to input: ncells, dist, k_joint; number of zinput
2. Create xcell matrix from inputs
3. **Graph_ctpt**: graphing the center-points of the unit cells from xcell
4. Loop command for adding z-height values at “zinput” number of locations
5. Output: ncells and xcell

LINK_LENGTH

1. Loop command
 - a. For all the link in the x-direction, calculate the length:

$$Lx = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

2. Loop command
 - a. For all the link in the y-direction, calculate the length:

$$Ly = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

GRAPH_CTPT

1. Drawing the horizontal lines
 - a. For all the center-points on every row, draw a horizontal line connecting any two points
2. Drawing the vertical lines
 - a. For all the center-points on every column, draw a vertical line connecting any two points

GUESS_INTERPOLATING

Step 1: using 'interp1' to draw a straight between all the Z inputs at the edge/border.

- 1) If ncells(2)>1: Determine there is more than one column in the matrix
 - i) **perform loop if matrix is more than a horizontal line.
 - b) find the index of all the centerpts in the 1st row that Z is inputted
 - c) Connect all the z-input in line(s) using 'interp1'
- 2) If ncells(2)>1;
 - a) find the index of all the centerpts in the last row that Z is inputted
 - b) Connect all the z-input in line(s) using 'interp1'
- 3) If ncells(1)>1: Determine there is more than one row in the matrix
 - a) find the index of all the centerpts in the 1st column that Z is inputted
 - b) Connect all the z-input in line(s) using 'interp1'
- 4) If ncells(1)>1;
 - a) find the index of all the centerpts in the last column that Z is inputted
 - b) Connect all the z-input in line(s) using 'interp1'

Step 2: after creating a border, create a second xcellvar called 'xcellvar2' that states those values at the borders are constrained

- 1) Creating the 'on/' 'off' matrix for controlling the constraints
 - a) Initiating an emptied matrix: xcellvar2
 - i) Note: xcellvar2 will be a matrix of the same size as xcells
 - ii) For any Cartesian coordinate variables fixed in xcell,
 - (1) The corresponding value in Xcellvar2=0
 - iii) For any Cartesian variables free to change in xcell,
 - (1) The corresponding value in Xcellvar 2=1

Step 3: using the Z values at the edge for each row, use 'interp1' to draw a straight line through all the Z inputs for each rows plus the z's at the edge.

Step 4: repeat process for the columns using 'interp1'

Step5: super-impose the z values derived from the row 'interp1' and the coln 'interp1'
Where there is overlap, take the max of the the Zvalues.

Step 6: Repeat step 2-5 to interpolate for the rows and column that does not have a zinputs.

Use the previously interpolated points as fixed values for xcellvar3.

MINIMIZINGENERGY

- 1) From the main page, Matrix30_interp, replacing all the non-constrained coordinate values in the xcell matrix with xi values: $xi = \{x(1), x(2), x(3) \dots\}$
- 2) Energy in the X direction
 - a) Sum-up all the energy between two center-points in the x-direction using atan2
- 3) Energy in the Y direction
 - a) Sum-up all the energy between two center-points in the y-direction using atan2
- 4) Potential Energy
 - a) Add y-direction sum to x-direction sum

CONTROLLING_LENGTH

- 1) Controlling the length in X direction
 - a) L_x : Calculating the distance between any two center-points in the x-direction
 - i) Subtract L_x from $dist^2$. This value should be as close as possible to zero or Fmincon would re-evaluate the Cartesian coordinates of the unit cells.
- 2) Controlling the length in Y direction, same process as above
 - a) L_y : Calculating the distance between any two center-points in the y-direction
 - i) Subtract L_y from $dist^2$. This value should be as close as possible to zero or Fmincon would re-evaluate the Cartesian coordinates of the unit cells.

BOUNDARY_VECTORS

adding extra unit cells for help calculating the unit vectors V and n.
Below is an ascii art diagram of the additional unit cells

```
% visual descriptions of boundary cells: xedge_top, xedge_bot, xedgeleft, xedge_right:

% =====xedge_top=====

% =xedge_left== [input xcell ct-pt ]==xedge_right==

% =xedge_left== [input xcell ct-pt ]==xedge_right==

% =====xedge_bot=====
```

- 1) Create the xedge_top, xedge_bot, xedgeleft, xedge_right, vectors
- 2) Add on these vectors to the xcell matrix

CAL_V_N

To understand the algorithm for this section, refer to the below ascii art diagram

numbering convention for cells:

```
% 4 5 6
% 1 2 3
```

numbering conventions for the v vectors and n vectors on each unit cells

3 $0,4 + 2$ 1	7 $8 + 6$ 5	11 $12 + 10$ 9
---------------------	-------------------	----------------------

These numbering conventions will help you understand how the code operates by starting with the first unit cell. Within each unit cells are 4 v and 4 n vectors. The code starts calculating at the first unit cell. Below is the algorithm for the code.

- 1) Calculating the unit $v_vectors$
 - a) $v_vector_i = (x_{cell_i} - x_{cell_{i+1}}) / (|(x_{cell_i} - x_{cell_{i+1}})|)$
- 2) Calculating the $n_vectors$
 - a) finding ϕ : the angle of rotation of each $n_vectors$: $(\sin(Z/L))^{-1}$
 - i) $L = (\text{length of the unit vector in the direction of either X or Y}) = 1$
 - b) finding the new rotated normal
 - i) for all $n_vectors$ in the plane with the horizontal V -vectors (e.g. #4 and #2 in the counting convention for individual unit cells)
 - ii) for all $n_vectors$ in the plane with the vertical V -vectors (e.g. #1 and #3 in the counting convention for individual unit cells)

CALC_THETA

- 1) Initialize the w_matrix , which is the a matrix consisting of all 8 $w_vectors$ before deformation
- 2) Calculate the $w_vectors$ after deformation
- 3) Calculate the side a vectors
- 4) Calculate the middle a vector
- 5) Calculate the u_vector
- 6) Calculate the θ s

BDPT_REMOVE

- 1) Initialize blank matrix
- 2) Start counting cells and storing cells at the first cell that is not on the border.
- 3) Skip the outer side cells
- 4) Stop counting when before reaching the first cell on the top border.

STICKFIGURE

```

close all;
clear all;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%INPUTS%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
global ncells;
global dist;
global k_joint;
global xcellvar;
global xcell_initial;
global std_dev;
global xcell;
global zinput;
[ncells,xcell] = user_inputs(ncells,dist,k_joint)
%function that ask for user inputs

xcell_initial=xcell; %storing the intital guess coordinates

std_dev= dist;
% this is is the deviation from the original guess
%for the upper bound. THE lower bound will always be '0'

%%creating the 'on/' 'off' matrix for what value can change
xcellvar=zeros(size(xcell));

counter31=0;
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1;
        if xcell(counter31,3)==0
            %if xcell=0 note: '0.001' is not '0'
            xcellvar(counter31,1)=1;
            %then put these 1 or 0 into xcellvar
            xcellvar(counter31,2)=1;
            xcellvar(counter31,3)=1;
        else
            xcellvar(counter31,1)=1;
            xcellvar(counter31,2)=1;
            xcellvar(counter31,3)=0;
        end
    end
end

end
xcellvar;

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% LT%    upper Lt 0  0  0  0  upperRT
% S%      0  0  0  0  0  0
% I%      0  0  0  0  0  0      ^
% D%      1stPT 0  0  0  0  LowerRT  | Y
% E
%      -----Bottom row-----
%      -----X->-----> X

```

```

xcellvar(1,:)= [0 0 0];
% nXm size of the xcell_bdpt
%constraint the first pt with all '0' b/c 1st pt is constraint

```

```

%Constraining the various coordinate at the corners of the matrix

```

```

xcellvar(ncells(2)*ncells(1)-ncells(2)+1,3)=0;
%constraint the upper LT Z coodinate
xcellvar(ncells(2)*ncells(1)-ncells(2)+1,1)=0;
%constraint the upper LT x coodinate
xcellvar(ncells(2),3)=0;
%constraint the lower RT Z coodinate
xcellvar(ncells(2),2)=0; %constraint the lower RT 1 coodinate

```

```

if ncells(1)==1
    xcellvar(:,2)=0;
end

```

```

if ncells(2)==1
    xcellvar(:,1)=0;
end

```

```

xcellvar(ncells(1)*ncells(2),3)=0;
%constraint the upper RT Z coodinate

```

```

[Lx,Ly] =link_length (ncells, xcell);
%function that finds the length of link:
%this will show what the intial guess gives

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Intital guesses for 3D%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

%putting all the intial guess in a vector
%except for the first coordinate
%as constraint and the input.

```

```

% Intital guess to be placed into fmincon
x0=[];

counter31=0;
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1;
        %% interiopr cell conunter Start at the 1st interior cell
        for coord=1:3
            if xcellvar(counter31,coord) ==1;
                %if xcell=0 note: '0.0000001' is not '0'
                x1=xcell(counter31,coord);
                x0=[x0;x1];
            end
        end
    end
end
x0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

graph_ctpt (ncells, xcell);
% a function that graphs the center pt

[Lx,Ly]=link_length (ncells, xcell);
%function that finds the length of link:
    %this will show what the intial guess gives

xcell;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Modified Intital guesses using fsolve and x0
%this modified initial guess will move all the points
    %to draw a straight line between two input points.
%the coordinates of the straight lines will give
    %the coordinates for the values for inputting into
    %fmincon for minmizing the potential energy ofthe system

tic;

%n_interp=(ncells(1)*ncells(2)-zinput)/2

%while n_interp>=ncells(1) & n_interp>=ncells(2)
xcell = guess_interpolating(xcell,xcellvar)

    %n_interp=n_interp/2
    %end

```

```

toc;
sec_elapsed_interpolation=toc

graph_ctpt (ncells, xcell);
%function that graphs the center points of unit cell

[Lx,Ly]=link_length (ncells, xcell);
%function that finds the length of link:
%this will show what the intial guess gives

xcell

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%putting all the intial guess in a vector
%except for the first coordinate as constraint and the input.
% Intial guess to be placed into fmincon
x0=[];

counter31=0;
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1;
        %% interiopr cell conunter Start at the 1st interior cell
        for coord=1:3
            if xcellvar(counter31,coord) ==1;
                %if xcell=0 note: '0.0000001' is not '0'
                x1=xcell(counter31,coord);
                x0=[x0;x1];
            end
        end
    end
end
x0;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

tic;

options=optimset('Display','iter','MaxIter',1e5,...
    'MaxFunEvals',1e12,'TolFun',0.001, 'TolX',0.001);

x = fmincon('minimizingEnergy',x0,[],[],[],[],0,[],'controlling_length',options);
%x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon)
%x = fmincon(fun,x0,A,b,Aeq,beq,lb,ub,nonlcon,options,P1,P2, ...)

```

```

    %subjects the minimization to the nonlinear inequalities c(x)
    % or equalities ceq(x) defined in nonlcon.
    %fmincon optimizes such that c(x) <= 0 and ceq(x) = 0.
    % Set lb=[] and/or ub=[] if no bounds exist.
    %FMINCON requires at least four input argument

toc;
sec_elapsed_fmincon_energymin=toc

xcell;
graph_ctpt (ncells, xcell)
[Lx,Ly]=link_length (ncells, xcell)
%function that finds the length of link: this will show what the intial guess gives

xcell
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%3D%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

graph_ctpt (ncells, xcell);

xcell %THE new xcell matrix with recalculated X and Y coordinates

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Graphing the center points%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

graph_ctpt (ncells, xcell)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Adding Boundary pts%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[xcell]=boundary_vectors (dist, ncells,xcell); %function that adds boundary pts

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Calculating the V vectors and norms%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[v_vectors,phi,rot_norm]= cal_v_n (ncells,xcell);

```

USER INPUTS

```

function [ncells,xcell,zinput] = user_inputs(ncells,dist,k_joint)

global ncells;
global dist;
global k_joint;
global xcellvar;
global xcell_initial;
global std_dev;
global xcell;
global zinput;

ncells= input('Enter the rows and columns in this form: [row, column]=');
dist=input('Enter distant between two center point=');
k_joint=input('Enter the joint stiffness value=');
% joint stiffness value
zinput=input('Enter the number Z heights that you will input=');

%creating a matrix of center points based upon input
xcell=[];
for it60=1:ncells(1)
    for it61=1:ncells(2)
        xcell2=[dist*(it61-1), dist*(it60-1),0];
        xcell=[xcell;xcell2];
    end
end
xcell;

graph_ctpt (ncells, xcell);
% a function that graphs the center pt

disp('-----')
repeatn=input('Do you want to modify values inputed? 1->yes and 0->no ==');
if repeatn==1;
    ncells= input('Enter the rows and columns in this form: [row, column]=');
    dist=input('Enter distant between two center point=');
    k_joint=input('Enter the joint stiffness value=');
    % joint stiffness value

    zinput=input('Enter the number Z heights that you will input=');
    disp('-----')

```

```

xcell=[];
for it60=1:ncells(1)
    for it61=1:ncells(2)
        xcell2=[dist*(it61-1), dist*(it60-1),0];
        xcell=[xcell;xcell2];
    end
end

graph_ctpt(ncells, xcell); % a function that graphs the center pt
repeatn=input('Do you want to modify values inputed? 1->yes and 0->no ==');

while repeatn==1;
    ncells= input('Enter the rows and columns in this form: [row, column]=');
    dist=input('Enter distant between two center point=');
    k_joint=input('Enter the joint stiffness value=');
    % joint stiffness value

    zinput=input('Enter the number Z heights that you will input=');
    disp('-----')

    xcell=[];
    for it60=1:ncells(1)
        for it61=1:ncells(2)
            xcell2=[dist*(it61-1), dist*(it60-1),0];
            xcell=[xcell;xcell2];
        end
    end
    graph_ctpt(ncells, xcell); % a function that graphs the center pt
    repeatn=input('Do you want to modify values inputed? 1->yes and 0->no ==');
end

%Adding the Z displacement for the number of zinput requested
for it62=1:zinput
    disp('-----')
    fprintf('Below are requests for the location and value for #%.0f from the total Z heights \n', it62 )
    deformz=input('Enter the [row,column] that you want to input the Z height=');%[row column];
    fprintf('Initial coordinates for the location requested: %.3f %.3f %.3f\n', ...
        xcell((deformz(1)-1)*ncells(2)+deformz(2), : ) )
    xcell((deformz(1)-1)*ncells(2)+deformz(2),3)=...
        input('Enter the Z height deformation for this point=');
end
graph_ctpt(ncells, xcell); % a function that graphs the center pt

```

```

disp('-----')

repeatz=input('Do you want to modify Z height inputs? 1->yes and 0->no ==');

if repeatz==1
    disp(' for statement below: only specify the number of Z heights to modify')
    zinput=input('Enter the number Z heights that you will input=');
    for it62=1:zinput

```


LINK_LENGTH

```
function [Lx,Ly]=link_length(ncells, xcell);

%Finding the initial magnitude of X and Y based on pythagorn theorem
Lx=[]; %initial length of based upon the X and Y and Z coordinates in X direction

count40=0;
for it42=1:ncells(1); %2
    for it40= 1:ncells(2)-1; % 1: 4-1=3
        Lx1=sqrt( (xcell(it40+count40,1)- xcell(it40+count40+1,1))^2 ...
            + (xcell(it40+count40,2)- xcell(it40+count40+1,2))^2 ...
            + (xcell(it40+count40,3)- xcell(it40+count40+1,3))^2);
        Lx=[Lx;Lx1];
    end
    count40=count40+ncells(2);
end
Lx;

Ly=[]; %initial length in the Y direction
count41=0;
for it44=1:ncells(1)-1
    for it45=1:ncells(2)
        Lyl=sqrt( (xcell(it45+count41,1)- xcell(it45+ncells(2)+count41,1))^2 ...
            + (xcell(it45+count41,2)- xcell(it45+ncells(2)+count41,2))^2 ...
            + (xcell(it45+count41,3)- xcell(it45+ncells(2)+count41,3))^2);
        Ly=[Ly;Lyl];
    end
    count41=count41+ncells(2);
end
Ly;
```

GRAPH_CTPT

```

function graph_ctpt (ncells, xcell);

figure;
%drawing the horizontal lines for the graph that connects two points

counter35=0;
for it35=1:ncells(1) % the numbers of rows
    for it36= 1:ncells(2)-1
        %%%2 1 less line than the total number of point per rows...
        %%%b/c # of lines connecting two points
        plot3([xcell(it36+counter35,1);xcell(it36+1+counter35,1)],...
            [xcell(it36+counter35,2);xcell(it36+1+counter35,2)],...
            [xcell(it36+counter35,3);xcell(it36+1+counter35,3)],'b*-');
        hold on
    end
    counter35=counter35+ncells(2);
end

%drawing the vertical lines for the graph that connects two points
counter36=0;
for it38= 1: ncells(1)-1 % numbers of column of lines
    for it37=1:ncells(2)
        plot3([xcell(it37+counter36,1);xcell(it37+ncells(2)+counter36,1)],...
            [xcell(it37+counter36,2);xcell(it37+ncells(2)+counter36,2)],...
            [xcell(it37+counter36,3);xcell(it37+ncells(2)+counter36,3)],'r*-');
        hold on
    end
    counter36=counter36+ncells(2); %3
end

title('Profile of position vector')
xlabel('X position')
ylabel('Y position')
zlabel('Z position')
axis equal

```

GUESS_INTERPOLATING

```
function xcell= guess_interpolating(xcell,xcellvar)

global ncells;
global xcell;
global dist;
global k_joint;
global xcellvar;
global xcell_initial;
global std_dev;
global zinput;

%step 1:
%using 'interp1' to draw a straight line between all the Z inputs at the edge/border
%using the X coordinates as reference.
%For reminder, all the corners Z values are constrained either at 0 ...
%%or at the value the user specify.

%step 2:
%after creating a border
%create a second xcellvar called 'xcellvar2' ...
%%that states those values at the borders are constrained.

%step 3:
%using the Z values at the edge for each row, use 'interp1' ...
%to draw a straight line through all the
%Z inputs for each row plus the z's at the edge.

%step 4:
%repeat process for the columns using 'interp1'

%step 5:
%super-impose the z values derived from the row 'interp1' and the column 'interp1'
% Where there is overlap, that the average of the Z values.
```

```

%%%%%%%%step 1:=%%%%%%%%step 1:=%%%%%%%%
if ncells(2)>1;
%find the index of all the centerpts in the 1st row that Z is inputed
vrow1= find(xcellvar(1:ncells(2),3)==0);
%Connect all the z-input in line(s)using 'interp1'
xcell(1:ncells(2),3)=INTERP1(xcell(vrow1,1),xcell(vrow1,3),xcell(1:ncells(2),1),'linear');
end

%find the index of all the centerpts in the last row that Z is inputed
if ncells(2)>1;
vrow2=[];
for it=ncells(2)*ncells(1)-ncells(2)+1:ncells(2)*ncells(1)
    if xcellvar(it,3)==0
        vrow2=[vrow2,it];
    end
end
%ranges= the centerpts for the last row
ranges=ncells(2)*ncells(1)-ncells(2)+1:ncells(2)*ncells(1);
%Connect all the z-input in line(s) using 'interp1'
xcell(ranges,3)=INTERP1(xcell(vrow2,1),xcell(vrow2,3),xcell(ranges,1),'linear');
end

%find the index of all the centerpts in the 1st column that Z is inputed
if ncells(1)>1;
vcoln1=[];
for it=1:ncells(2):ncells(2)*ncells(1)-ncells(2)+1;
    if xcellvar(it,3)==0
        vcoln1=[vcoln1,it];
    end
end
%ranges= the centerpts for the 1st column
ranges=1:ncells(2):ncells(2)*ncells(1)-ncells(2)+1;
%Connect all the z-input in line(s) using 'interp1'
xcell(ranges,3)=INTERP1(xcell(vcoln1,2),xcell(vcoln1,3),xcell(ranges,2),'linear');
end

%find the index of all the centerpts in the last column that Z is inputed
if ncells(1)>1;
vcoln2=[];
for it=ncells(2):ncells(2):ncells(2)*ncells(1);
    if xcellvar(it,3)==0
        vcoln2=[vcoln2,it];
    end
end
%ranges= the centerpts for the last column
ranges=ncells(2):ncells(2):ncells(2)*ncells(1);

```

```

%Connect all the z-input in line(s) using 'interp1'
xcell(ranges,3)=INTERP1(xcell(vcoln2,2),xcell(vcoln2,3),xcell(ranges,2),'linear');
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%step 2:=%%%%%%%%%%step 2:=%%%%%%%%%%
% Creating second matrix of 'xcellvar2'
xcellvar2=zeros(size(xcell));

counter31=0;
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1;
        if xcell(counter31,3)==0 %if xcell=0 note: '0.001' is not '0'
            xcellvar2(counter31,1)=1; %then put these 1 or 0 into xcellvar
            xcellvar2(counter31,2)=1;
            xcellvar2(counter31,3)=1;
        else
            xcellvar2(counter31,1)=1;
            xcellvar2(counter31,2)=1;
            xcellvar2(counter31,3)=0;
        end
    end
end

end

% constraining the Z at the LT side
if ncells(1)>1
    for it31= ncells(2)+1:ncells(2):ncells(2)*ncells(1)-ncells(2)+1;
        xcellvar2(it31,3)=0;
    end
end

% constraining the Z at the RT side
if ncells(1)>1
    for it31= ncells(2):ncells(2):ncells(2)*ncells(1)
        xcellvar2(it31,3)=0;
    end
end

% constraining the Z on Bottom row an
if ncells(2)>1
    for it30=1:ncells(2)
        xcellvar2(it30,3)=0; %constraint the bottom row z coordinate pt
    end
end
end

```

```

% constraining the Z on Top row an
if ncells(2)>1
    for it30=ncells(2)*ncells(1)-ncells(2)+1:ncells(2)*ncells(1)
        xcellvar2(it30,3)=0; %constraint the bottom row z coordinate pt
    end
end
xcellvar2;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%step 3 and 4:=%%%%%%%%%%step 3and4:=%%%%%%%%%% %%%%%%%%%%

%Drawing straight through each row
if ncells(2)>1
    xcellr=[];
    for it=1:ncells(2):ncells(2)*ncells(1)-ncells(2)+1;
        irows=[];
        for it2=it:it+ncells(2)-1;
            if xcellvar2(it2,3)==0;
                irows=[irows; it2];
            end
        end
        rrange=it:it+ncells(2)-1;
        xcellr(rrange,3)=INTERP1(xcell(irows,1),xcell(irows,3),xcell(rrange,1),'linear');
    end
else
    xcellr=zeros(size(xcell));
end

%drawing a straight line through each column
if ncells(1)>1
    xcellc=[];
    for it= 1:ncells(2);
        icoln=[];
        for it2=it:ncells(2):ncells(1)*ncells(2);
            if xcellvar2(it2,3)==0;
                icoln=[icoln;it2];
            end
        end
        crange=it:ncells(2):ncells(1)*ncells(2);
        xcellc(crange,3)=INTERP1(xcell(icoln,2),xcell(icoln,3),xcell(crange,2),'linear');
    end
else
    xcellc=zeros(size(xcell));
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%step 5:=%%%%%%%% %step 5:=% %%%%%%%%%%%%%%%

xcellc;
xcellr;

if ncells(2)>1 & ncells(1)>1

    xcell(:,3)=max (xcellc(:,3),xcellr(:,3));

elseif ncells(2)>1 & ncells(1)==1
    xcell(:,3)= xcellr(:,3);
else
    xcell(:,3)= xcellc(:,3);
end

xcell;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%REPEATING STEP 2-5%%%%%%%%REPEATING STEP 2-5%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%step 2:=%%%%%%%%step 2:=%%%%%%%%

% Creating third matrix of 'xcellvar3'
xcellvar3=zeros(size(xcell));

counter31=0;
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1;
        if xcell(counter31,3) ==0 %if xcell=0 note: '0.0000001' is not '0'
            xcellvar3(counter31,1)=1; %then put these 1 or 0 into xcellvar
            xcellvar3(counter31,2)=1;
            xcellvar3(counter31,3)=1;
        else
            xcellvar3(counter31,1)=1;
            xcellvar3(counter31,2)=1;
            xcellvar3(counter31,3)=0;
        end
    end
end

end
% constraining the Z at the LT side
if ncells(1)>1
    for it31= ncells(2)+1:ncells(2):ncells(2)*ncells(1)-ncells(2)+1;
        xcellvar3(it31,3)=0;
    end
end
end

```

```

% constraining the Z at the RT side
if ncells(1)>1
    for it31= ncells(2):ncells(2):ncells(2)*ncells(1)
        xcellvar3(it31,3)=0;
    end
end

% constraining the Z on Bottom row an
if ncells(2)>1
    for it30=1:ncells(2);
        xcellvar3(it30,3)=0; %constraint the bottom row z coordinate pt
    end
end

% constraining the Z on Bottom row an
if ncells(2)>1
    for it30=ncells(2)*ncells(1)-ncells(2)+1:ncells(2)*ncells(1)
        xcellvar3(it30,3)=0; %constraint the bottom row z coordinate pt
    end
end
xcellvar3;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% End step Two%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%step 3 and 4:=% %%%%%%%%%step 3and 4:=% %%%%%%%%%

%Drawing straight through each row
if ncells(2)>1
    xcellr=[];
    for it=1:ncells(2):ncells(2)*ncells(1)-ncells(2)+1;
        irows=[];
        for it2=it:it+ncells(2)-1;
            if xcellvar3(it2,3)==0;
                irows=[irows; it2];
            end
        end
        rrange=it:it+ncells(2)-1;
        xcellr(rrange,3)=INTERP1(xcell(irows,1),xcell(irows,3),xcell(rrange,1),'linear');
    end

end

%drawing a straight line through each column
if ncells(1)>1
    xcelle=[];
    for it= 1:ncells(2);

```



```

icoln=[];
for it2=it:ncells(2):ncells(1)*ncells(2);
    if xcellvar3(it2,3)==0;
        icoln=[icoln;it2];
    end
end
crange=it:ncells(2):ncells(1)*ncells(2);
xcellc(crange,3)=INTERP1(xcell(icoln,2),xcell(icoln,3),xcell(crange,2),'linear');
end

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%step 5:=% %%%%%%%%%%%step 5:=%%%%%%%%%%

xcellc;
xcellr;

if ncells(2)>1 & ncells(1)>1
    xcell(:,3)=max (xcellc(:,3),xcellr(:,3));
elseif ncells(2)>1 & ncells(1)==1
    xcell(:,3)= xcellr(:,3);
else
    xcell(:,3)= xcellc(:,3);
end

xcell;

```

MINIMIZINGENERGY

```
function pot_energy = minimizingEnergy(x)
```

```
global ncells;
global xcell;
global dist;
global k_joint;
global xcellvar;
global xcell_initial;
global std_dev;
```

```
%replacing all the non-constrained coordinate values with xi values...
```

```
%%These are the unknowns
```

```
cnt = 1; % counter for the x(1), x(2), x(3) etc...
```

```
counter31=0;
```

```
for it30= 1: ncells(1);% % rows
```

```
    for it31= 1:ncells(2)% % column
```

```
        counter31=counter31+1;
```

```
        for coord=1:3
```

```
            if xcellvar(counter31,coord) ==1
```

```
                %if xcell=0 note: '0.001' is not '0'
```

```
                xcell(counter31,coord)=x(cnt);
```

```
                cnt = cnt + 1;
```

```
            end
```

```
        end
```

```
    end
```

```
end
```

```
%-----0-----
```

```
%      ^  \ )A
```

```
%      /   \
```

```
%      /     v
```

```
%      / )C ----0-----
```

```
%--0-----
```

```
%energy in the X direction
```

```
pot_energy=0;
```

```
%initial length of based upon the X and Y and Z coordinates in Xdirection
```

```
if ncells(2)>2
```

```

counter31=2;
for it42=1:ncells(1); %row
    for it40= 1:ncells(2)-2; % column
        counter31=counter31+1;
        angle= -pi ...
            +(atan2(xcell(counter31,3)-xcell(counter31-1,3),...
                xcell(counter31,1)-xcell(counter31-1,1)))... %A
            -atan2(xcell(counter31-1,3)-xcell(counter31-2,3), ...
                xcell(counter31-1,1)-xcell(counter31-2,1)) ; %C

        pot_energy = pot_energy + 0.5 *k_joint *(pi-abs(angle))^2;

    end
    counter31=counter31+2;

end
end

%energy in the Y direction
if ncells(1) > 2
    counter31=2;
    for it44=1:ncells(1)-2;%row
        for it45=1:ncells(2)%column
            counter31=counter31+1;
            angle= -pi ...
                +(atan2(xcell(counter31+2*ncells(2)-2,3)-xcell(counter31+ncells(2)-2,3),...
                    xcell(counter31+2*ncells(2)-2,2)-xcell(counter31+ncells(2)-2,2)))... %A
                -atan2(xcell(counter31+ncells(2)-2,3)-xcell(counter31-2,3),...
                    xcell(counter31+ncells(2)-2,2)-xcell(counter31-2,2)) ; %C

            pot_energy = pot_energy + 0.5 *k_joint *(pi-abs(angle))^2;

        end
    end
end

pot_energy;

```

CONTROLLING_LENGTH

```

function [c,lengths] = controlling_length(x)

%nonlcon
%The function that computes the nonlinear inequality constraints c(x)<= 0
%and the nonlinear equality constraints ceq(x) = 0.
%The function nonlcon accepts a vector x and returns two vectors c and ceq.
%The vector c contains the nonlinear inequalities evaluated at x,
%and ceq contains the nonlinear equalities evaluated at x.
%The function nonlcon can be specified as a function handle.

%x = fmincon(@myfun,x0,A,b,Aeq,beq,lb,ub,@mycon)

%where mycon is a MATLAB function such as

%function [c,ceq] = mycon(x)
%c = ...    % Compute nonlinear inequalities at x.
%ceq = ...  % Compute nonlinear equalities at x.

global dist;
global ncells;
global k_joint;
global xcellvar;
global xcell_initial;
global std_dev;
global xcell;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Keeping the link length constant%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Controlling the length in X direction
Lx=[]; %initial length of based upon the X and Y and Z coordinates in Xdirection
%counter31=ncells(2)+2+1; % interiopr cell conunter Start at the 1st interior cell
counter31=0;
for it42=1:ncells(1); %row
    for it40= 1:ncells(2)-1; % column
        counter31=counter31+1; %% interiopr cell conunter
        Lx1=(xcell(counter31,1)- xcell(counter31+1,1))^2 ...
            + (xcell(counter31,2)- xcell(counter31+1,2))^2 ...
            + (xcell(counter31,3)- xcell(counter31+1,3))^2- dist^2;
        Lx=[Lx;Lx1];
    end
    counter31=counter31+1;
end
end

```

```

%Controlling the length in Y direction
Ly=[];
%counter31=ncells(2)+2+1;% interiopr cell conunter Start at the 1st interior cell
counter31=0;
for it44=1:ncells(1)-1%row
    for it45=1:ncells(2)%column
        counter31=counter31+1; %% interiopr cell conunter
        Lyl=(xcell(counter31,1)- xcell(counter31+ncells(2),1))^2 ...
            + (xcell(counter31,2)- xcell(counter31+ncells(2),2))^2 ...
            + (xcell(counter31,3)- xcell(counter31+ncells(2),3))^2- dist^2;
        Ly=[Ly;Lyl];
    end
    % counter31=counter31+2;
end

lengths=[Lx;Ly];

%%%%%%%%all X(i) values should be positive and within a range of std %%%%%%%%%

% due to the nonlinear inequality constraint of fmincon c(x)<= 0.
%therefore all the neg of x(i) [-x(i)] values should be less than or equal to 0...
%This means that all the x(i) must be positive.
if 0 %if 0=false means will notever do what to do within if loop.
    %%if=1, then true, do everything in the loop
    values=[];
    counter31=0;
    cnt=1;
    for it30= 1: ncells(1);% % rows
        for it31= 1:ncells(2)% % column
            counter31=counter31+1; %% interiopr cell conunter Start at the 1st interior cell
            for coord =1:3

                if xcellvar(counter31,coord) ==1; %if xcell=0  note: '0.0000001' is not '0'

                    if coord==3
                        LBZ= -x(cnt);
                        %lower bound for the z coodinate which is the lowest value inputed
                        %from this equation:  $0 \leq x(i) \leq x(i) + \text{std\_dev}$ .
                        % Solve for x(i) for the LT .
                        UBZ= x(cnt)- ( max(xcell_initial(:,3))+dist/2);
                        %upper bound for the z coodinate which is the highest value inputed
                        values=[values;LBZ;UBZ];
                    else
                        LB= - x(cnt); % this sets up the LB to be zero.
                        %from this equation:  $0 \leq x(i) \leq x(i) + \text{std\_dev}$ .

```

```

    %Solve for x(i) for the LT .
    UB= x(cnt) - (xcell_initial(counter31,coord)+std_dev);

    %upper bound for the the X and Y coodinate...
    %which the +3 pts away original values from this equation:
    % 0 <= x(i) <= x(i) + std_dev. Solve for x(i) for the RT side
    values=[values;LB;UB];

    end

    cnt=cnt+1;
    end
    end
    end
    end
    c=values;
else
    c=0;
end
end

```

BOUNDARY_VECTORS

```
function [xcell]= boundary_vectors (dist, ncells, xcell );

global ncells;
global xcell;
global dist;
global k_joint;
global xcellvar;
global xcell_bdpt; %intial guess xcell with boundary pts
global xcell_initial;
global std_dev;

%The matrix is floating.
%These boundary vectors are added to help find the position v_vectors and the n_vector
%because of too much constraint. For the design of this crust,..
%a flexible skin will be over the crust.
% this Skin will attach the crust to the base and stretches while the crust deforms.
%Presently, the skin is not developed yet.
%Ex: A piece of cloth on a square piece of stretchy rubber...
%The edge of the rubber is attached a frame.
%The cloth can deform in various ways within the limits of the rubber.
%Therefore the edge of the matrix is not attached, but there are constraints.
%The constraints that this code will consider are
%the coordinate of first point which will be from the user input
%Another constraints to prevent rotation is the X value of Centerpt ..
%from 1 of the neighboring unit cell

% visual descriptions of boundary vectors of
%xedge_top, xedge_bot, xedgeleft, xedge_right:

% =====xedge_top=====

% =xedge_left=== [input xcell ct-pt ]===xedge_right===

% =xedge_left=== [input xcell ct-pt ]===xedge_right===

% =====xedge_bot=====

xedge_bot = xcell(1:ncells(2),:); %rosen's simplification of my coding from 13 lines to 2
xedge_bot(:,2) = xedge_bot(:,2) - dist;
xedge_bot=[0,0,0;xedge_bot; 0,0,0];
%the (0,0,0) are placement pts. Are ignored/skip over during calculation

xedge_top =xcell( (ncells(2)*(ncells(1)-1)+1):ncells(2)*ncells(1),: );
```

```

xedge_top(:,2) = xedge_top(:,2) + dist;
xedge_top=[0,0,0;xedge_top; 0,0,0];

%We need an extra set of vectors at the left edge of the first column...
%in the matrix to define the edge of the crust
xedge_left=[];
for it12= 1:ncells(2):(ncells(2)*(ncells(1)-1)+1);
    xedge_left1=xcell(it12,:);
    xedge_left1(:,1)=xedge_left1(1)- (dist);
    %xedge_left2=[xedge_left1(1)- (dist/2), xedge_left1(2),xedge_left1(3)];
    xedge_left=[xedge_left; xedge_left1];
end

%We need an extra set of vectors at the right edge of the first column

xedge_right=[];

for it13= ncells(2):ncells(2):ncells(2)*ncells(1);
    xedge_right1=xcell(it13,:);
    xedge_right1(:,1)=xedge_right1(1)+ (dist);
    %xedge_right2=[xedge_right1(1)+ (dist/2), xedge_right1(2),xedge_right1(3)];
    xedge_right=[xedge_right; xedge_right1];
end

%combining both the inputted ct-pt of cells and boundary vectors as one matrix
all_xcells=[xedge_bot];
count17=0;
count18=0;
for it16=1:ncells(1)
    count18=count18+1;
    row1=[xedge_left(count18,:);xcell(it16+count17:
ncells(2)*it16,:);xedge_right(count18,:)];
    all_xcells=[all_xcells;row1];
    count17=count17+ncells(2)-1;
end

all_xcells=[all_xcells;xedge_top];
xcell=all_xcells;

```


CAL_V_N

```
function [v_vectors,phi,rot_norm] = cal_v_n(ncells,xcell)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Calculating the v vectors%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%numbering convention for cells:
% 4 5 6
% 1 2 3

% numbering convetions for the v vectors and normals"
% 3
%4 + 2
% 1

counter30=0; % location placement for calc. values of v
counter31=ncells(2)+2+1;% interiopr cell conunter

for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1; %
        counter30=counter30+1; %
        v_vectors(counter30,1,:)=(xcell(counter31-(ncells(2)+2),:)- xcell(counter31,:))...
            /(norm((xcell(counter31-(ncells(2)+2),:)- xcell(counter31,:))));

        v_vectors(counter30,2,:)=(xcell(counter31+1,:)- xcell(counter31,:))...
            /(norm(xcell(counter31+1,:)- xcell(counter31,:)));

        v_vectors(counter30,3,:)=(xcell(counter31+(ncells(2)+2),:)- xcell(counter31,:))...
            /(norm((xcell(counter31+(ncells(2)+2),:)- xcell(counter31,:))));

        v_vectors(counter30,4,:)=(xcell(counter31-1,:)- xcell(counter31,:))...
            /(norm(xcell(counter31-1,:)- xcell(counter31,:)));

    end
    counter31=counter31+2;

end
v_vectors;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Calculating the Normals with local UCS%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% SEE Notes and Drawings for more Explanation

%step 1: finding f=phi: the angle of rotation: (sin(Z/L))^-1
%Where L=(length of the vector in the direction of either X or Y)=1 ...
%becuase if unit vector

%Step 2: finding the new rotated normal
% for finding the Normals from the Horizontal V-vectors,...
%the Normal is rotated around the Y-axis,
% therefore the value of the Y coordinate is always '0' for H V-vectors
% from Jerry H. Ginsberg "Advanced Engineering Dynamic 2nd ed"
%Rotation matix for rotation about the Y-axis:
%[ cos(f) 0 -sin (f);
%  0  1  0;
% sin(f) 0 cos (f)];

% for finding the Normals from the Vertical V-vectors,...
%the Normal is rotated around the X-axis,
% therefore the value of the X coordinate is always '0' for V V-vectors
%[ 1  0  0;
%  0  cos(f) sin (f);
%  0 -sin(f) cos (f)];

%Step 3: use check equation to validate the new norm
%check EQN 1: dot(N2, N1)= norm(N2)* norm(N1)*cos(f)
%check EQN 2: dot(N2, V2)=0 because perpendicular
% N2= New Rotated Normal N1= Orginal Normal (0,0,1)
% V2= New Roated V Vector
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Step 1: find phi: angle of rotation%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% numbering conventions for the phi's
% 3_v
%4_h + 2_h
% 1_v

phi=[];
for it15= 1:ncells(1)*ncells(2)

    phiy=asin(v_vectors(it15,1,3)/(-1));
    % Calculating phi from Vertical V-vectors
    phi=[phi;phiy];

    phix=asin(v_vectors(it15,2,3)/1);
    % Calculating phi from horizontal V-vectors

```

```

phi=[phi;phix];

phiy2=asin(v_vectors(it15,3,3)/1);
% Calculating phi from Vertical V-vectors
phi=[phi;phiy2];

phix2=asin(v_vectors(it15,4,3)/(-1));
% Calculating phi from horizontal V-vectors
phi=[phi;phix2];
end
phi;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Step 2: finding the new rotated normal%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%for Horizontal V-vectors
%rot_maty=[ cos(f) 0 -sin (f);
%          0 1 0;
%          sin(f) 0 cos (f)];

%For Vertical V-vectors
%rot_matx= [ 1 0 0;
%           0 cos(f) sin (f);
%           0 -sin(f) cos (f)];

normal=[0,0,1];

rot_norm=[];

for it16=1:2:4*ncells(1)*ncells(2)
    rot_normx= [1, 0, 0; 0, cos(phi(it16)), sin(phi(it16));...
               0, -sin(phi(it16)), cos(phi(it16))]*normal';
    rot_norm=[rot_norm;rot_normx'];

    rot_normy= [cos(phi(it16+1)), 0, -sin(phi(it16+1)); 0, 1, 0;...
               sin(phi(it16+1)), 0, cos(phi(it16+1))]*normal';
    rot_norm=[rot_norm;rot_normy'];

end
rot_norm; % this matrix is already normalize need not to be normed

%For rot_norm, Place all the X in one matrix,...
%All the Y in another, Z in the last

for k = 1:ncells(1)*ncells(2)
    for k2 = 1:4

```

```

        for xyz = 1:3
            rot_norm3D(k,k2,xyz) = rot_norm((k-1)*4+k2,xyz);
        end
    end
end

for k = 1:ncells(1)*ncells(2)
    fprintf('Element %d of rot_norm3D\n', k);
    for kk = 1:4
        fprintf(' %f %f %f\n', rot_norm3D(k,kk, :));
    end
end

rot_norm3D;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Step 3: Validating the new rotated normal%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%plotting to check perpendicularity of 1 vector and 1 norm
%figure
%axis equal
%plot([0;v_vectors(1,3,2)],[0;v_vectors(1,3,3)],'c+-')
%hold on
%plot([0;rot_norm(3,2)],[0;rot_norm(3,3)],'b+-')
%axis equal

check_norm1=[]; % the results should all be '0'
check_norm2=[]; % the results should all be '0'

for it17=1:4*ncells(1)*ncells(2)
    check_eqn1= ( norm(rot_norm(it17,:))*cos(phi(it17)) )...
        - ( dot(rot_norm(it17,:),normal) );
    check_norm1=[check_norm1;check_eqn1];
end

count30=0;
for it18=1:ncells(1)*ncells(2)
    for it19=1:4
        count30=count30+1;
        check_eqn2=dot(rot_norm(count30,:),...
            [v_vectors(it18,it19,1),...
            v_vectors(it18,it19,2),...
            v_vectors(it18,it19,3)] );
        check_norm2=[check_norm2;check_eqn2];
    end
end
end

```

CALC_THETA

```
function [theta_tab] = calc_theta (ncells,xcell,v_vectors,rot_norm,phi)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Calculation using Paul's Equation for the first cell%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global k_alpha;
global k_beta;
global k_lambda;

%h_scaling is a scaling factor multiplied to the side of the small triangle
% Due to some re-calculation, Paul's equation was modified:
% w_norm=sin (B/2) => w_norm= (h_scaling)*sin (B/2)= tan (B/2)
%since h_scaling= 1/cos

h_scaling=1/(cos((k_beta)/2));
w_norm=tan((k_beta)/2);

%These are the initial vectors of 1 unit cell before any rotation.
w_unit=[-1, 0,0;
        1, 0,0;
        0,-1,0;
        0, 1,0;
        1, 0,0;
        -1, 0,0;
        0, 1,0;
        0,-1,0];
w_initial=[-0.1944, 0, 0;
           0.1944, 0, 0;
           0, -0.1944, 0;
           0, 0.1944, 0;
           0.1944, 0, 0;
           -0.1944, 0, 0;
           0, 0.1944, 0;
           0, -0.1944, 0];

w=[];
% w=example for 1 unit cell:[w8; w1; w2; w3; w4; w5; w6; w7] ...
%in reference to numbering convention per unit cell
count19=0;
for it20=1:ncells(1)*ncells(2)
    for it21=1:4
```

```

count19=count19+1;
w_vec1= (w_norm)*cross([v_vectors(it20,it21,1),v_vectors(it20,it21,2),...
    v_vectors(it20,it21,3)], rot_norm(count19,:));
w=[w;w_vec1]; %method of stacking **must start with w=[] for an emptied set
w_vec2= (w_norm)*cross(rot_norm(count19,:),[v_vectors(it20,it21,1),...
    v_vectors(it20,it21,2),v_vectors(it20,it21,3)]);
w=[w;w_vec2];
end
end
w;

```

a=[]; %a=side vector of top small connecting triangles

```

count=0;
for it20=1:ncells(1)*ncells(2);
    for it21=1:4
        count=count+1;
        a_val1=([v_vectors(it20,it21,1),v_vectors(it20,it21,2),...
            v_vectors(it20,it21,3)]+w(count,:))/(h_scaling);
        a=[a;a_val1];
        count=count+1;
        a_val2=([v_vectors(it20,it21,1),v_vectors(it20,it21,2),...
            v_vectors(it20,it21,3)]+w(count,:))/(h_scaling);
        a=[a;a_val2];
    end
end

```

a ; %example for 1 unit cell: a=[a8; a1; a2; a3; a4; a5; a6; a7]...

%all the vectors on the side of the main link triangle

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%find the middle a vector between two big link
%explanation of modification of paul's eqn for "for loop"

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

%a=example for 1 unit cell:[a8; a1; a2; a3; a4; a5; a6; a7]
%rearrange the a matrix to move the first a value from each unit cell...
%to become the last value for each unit cell
a2=[]; %a1 %a2 %a3....%a8

```

```

for it23= 1:8*ncells(1)*ncells(2)-7
    a1=[a(it23+1:it23+7,:);a(it23,:)];

```

```

    a2=[a2;a1];
end
a=a2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Reference Word Doc: findingM_a_mid_vector %%%%%%%%%%

m=[];

count4=0;
%for changing the vector 1 and vector2 from the 'a' matrix for 'For Loop it6'
%finding a2z=az_mid1 by searching through trial and error: in 'For Loop it5'
L1= 73/256; %length of the side of intermediate joints
L5=83/256; %length of the middle vector of intermediate joints
for it6=1:4*ncells(1)*ncells(2) % the total numbers of a_mid for all cells
    count4=count4+1;
    vect1=a(count4,:);
    count4=count4+1;
    vect2=a(count4,:);

    p_vect=0.25*vect2 - 0.25*vect1;
    %%%% p_vect= a vector that connects a(i) and a(i+1).
    q_vect=0.25*vect1 + p_vect/2;
    %%%%q_vect= vector taht connects the centerpt of unit cell to mid of p_vect.
    L4= sqrt(L1^2 - (norm(p_vect/2))^2);
    %%% length that connects Q_vector to the bottom of the triangle
    angle_triangle= acos( (-(L4^2)+(norm(q_vect))^2 +L5^2)...
        /(2*norm(q_vect)*L5));
    L6= norm(q_vect)*tan(angle_triangle);
    %%%L6= perpendicular from end of q_vector to L5
    n_avects= cross(vect1,vect2);
    %%%% normal vector to 2 crossing a vectors
    n_avectu= n_avects/(norm(n_avects));
    %%%% the unit vector of the a vector normal
    m_vect= -L6*n_avectu+q_vect;
    m_vectu= m_vect/(norm(m_vect));
    m=[m;m_vectu];
end

a_mid=m; % a matrix of all the calculated middle 'a' vectors...
        %%%between the intermidiate links.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%findin the U vectors that are normal to the intermediate links...

```

```

    %%%%using the 'a' and 'a_mid' vectors
%a=example for 1 unit cell:[ a1; a2; a3; a4; a5; a6; a7; a8]
%a_mid= [a_mid1(between a1 and a2); a_mid2(between a3 and a4);...
%%a_mid3(between a5 and a6); a_mid4(between a7 and a8)]

u_norm=[];
count5=0;
for it7=1:4*ncells(1)*ncells(2)
    count5=count5+1;
    u_val1=( cross(a2(count5,:), a_mid(it7,:)) )/...
        ( norm (cross(a2(count5,:), a_mid(it7,:)) ));
    u_norm=[u_norm;u_val1];

    count5=count5+1;
    u_val2= ( cross(a_mid(it7,:), a2(count5,:)) )/...
        ( norm (cross(a_mid(it7,:), a2(count5,:)) ));
    u_norm=[u_norm;u_val2];
end
u_norm;

%example for 1 unit cell:[8 rows, 3 colns]
    %%%%[ u1; u2; u3; u4; u5; u6; u7; u8]

%%%%%%%%%%explanation of finding theta%%%%%%%%%%
% find theta1 (between main and intermediate),
%theta2 (between 2 intermediate), theta3 (between main and intermediate)
% g_n= [g_n1; g_n2; g_n3; g_n4];
% k_lambda= [k_lambda1, k_lambda2, k_lambda3, k_lambda4];
% from paul's eqns:
% theta1=- acos(dot(u1,n1)+pi-lambda1
% theta3= -acos(dot(u2,n2)+pi-lambda2    modified from Paul's eqn: '+' => '-'.
% theta 2= -acos(u1,u2)+ pi
%%%%%%%%%%

theta=[]; % for every 3 rows are the 3 thetas for 1 out of 4 section...
    %%%of the radial symmetric unit cell.
%copying the beginning the 1st norm to the end for: ...
    %%'For loop it8' because of circular motion
g_n_it8= [rot_norm(1:4*ncells(1)*ncells(2),:); rot_norm(1,:)];
count6=0;
for it8= 1:4*ncells(1)*ncells(2)
    count6=count6+1;
    theta1= -acos( dot(u_norm(count6,:), g_n_it8(it8,:)) ) + pi - k_lambda;
    theta=[theta;theta1];

    theta2= -acos( dot(u_norm(count6,:), u_norm(count6+1,:)) ) +pi;

```



```

theta=[theta;theta2];

theta3= -acos( dot(u_norm(count6+1,:), g_n_it8(it8+1,:)) ) + pi - k_lamda;
theta=[theta;theta3];
count6=count6+1;

end
theta;

theta_tab=[];
for it24=1:12:3*4*ncells(1)*ncells(2)
    theta_t=[theta(it24:it24+11)];
    theta_tab=[theta_tab,theta_t];
end

theta_tab
%the Theta for each section of the first unit cell

```

BDPT_REMOVE

```
function xcell = bdpt_remove(ncells,xcell);
%removing the boundary pts/vectors for display for other purpose

xcell0=xcell; %storing the xcell with the boundary pts in xcell0

xcell_ctpt=[];
counter31=ncells(2)+2+1;% interiopr cell conunter Start at the 1st interior cell
% ncells(2)+2+1+1 = ncells(2)= bottom row...
    %%2= the 2 corners empty boundary cell 1=LT boundary cell
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1; %% interiopr cell conunter
        xcell_ctpt= [xcell_ctpt; xcell(counter31,:)];
        %removing the begin_pt and end_pt for graphings
    end
    counter31=counter31+2;
end

xcell=xcell_ctpt

%xcell=xcell0;
```

APPENDIX D:

MATLAB FOR METHOD 2: FIX FACE

Method 2 with one of the faces being fixed was also implemented through using MATLAB Coding. The first page will be a guide to the code and the functions in the code. The rest of the pages in this Appendix are the code.

NAVIGATION OUTLINE SECOND PROGRAM IMPLEMENTING METHOD 2: FIXED FACE

Nomenclature:

k1= stiffness of type-1 joint
k2= stiffness of type-2 joint
k_alpha= geometric design constant
k_beta = geometric design constant
k_lambda= geometric design constant
h_scaling= geometric design constant
xcell: a matrix of the Cartesian coordinates of the center-points of the unit cells
ncells: the matrix dimension
xcellvar: controlling the constraints. Dimension size is the same as xcell.
dist: fixed distance value between any two point
xcell_initial: initial value of xcell before the modifying any values.
zinputs: user dislocation inputs for the z-values in xcell

FIXFACE

- 1) Specifying inputs
- 2) k1,k2,dist,ncells,xcell, k_alpha, k_beta, h_scaling
- 3) Constraining certain variables
- 4) Xcellvar: “on/off” matrix for controlling what value from xcell matrix can change
- 5) xcell: an (m*n)-by-3 matrix, where x is in the first column, y is 2nd, z is 3th
- 6) 0= off, meaning value should remain fix
- 7) 1=on, value can change
- 8) Developing the initial guess
 - a) **cc_convert_scfix:**
 - i) Note: for all of the steps, skip the 4th n and v vectors, which are on 4th face of the first cell.
 - ii) Converting the Cartesian coordinates to Spherical
 - iii) Interpolating by rotating the S.C
 - iv) Condensing the variables by realization of duplications
 - b) Creating initial S.C. variables from previous function results:
 - i) phi_ni, theta_ni, theta_vi

- c) Creating x0: initial guess vector (Do NOT include the 4th n and v vectors)
- 9) Creating upper and lower bounds for x0
 - a) Lower bound for phi_ni, theta_ni, theta_vi
 - b) Upper bound for phi_ni, theta_ni, theta_vi

10) sc_graphfix(x) Function that graphs the unit cells using SC as inputs

11) Fmincon –blackbox MATLAB function

- a) **minimizingPenergyfix**: calculating the potential energy in the system for Fmincon

12) Expands the condensed matrices of S.C. coordinates vectors

- a) Creates relationship among vectors based upon duplications
- b) Uncondensing the variables: phi_n, theta_n, theta_v
- c) Reminder:
 - i) Numbering conventions for the v vectors and n vectors on each unit cells
 - ii) Numbering in reference in face number

3	7	11
4 + 2	8 + 6	12 + 10
1	5	9

- d) For all the sections in each cell in a matrix is m-by-n matrix

- i) One unit cell has 4 sections.

ii) phi_n

- (1) Skip the 4th n vectors on the 4th face of the 1st unit cell
- (2) if current variable is a multiple of 4 and is greater than 4: 8,12,14,...
 - (a) Copy the 6th previous phi_n variables
 - (b) e.g: phi_n at face number 12= phi_n at face number 6
 - (c) If not a multiple of 4, then pull values from the iteration results: x

iii) Theta_n

- (1) Skip the 4th n vectors on the 4th face of the 1st unit cell
- (2) If current variable is a multiple of 4 and is greater than 4: 8,12,14, ...
 - (a) Copy the 6th previous theta_n variables
 - (b) e.g: theta_n at face number 12 = theta_n at face number 6
 - (c) If not a multiple of 4, the pull values from the iteration results: x

iv) Theta_v

- (1) Skip the 4th v vectors on the 4th face of the 1st unit cell
- (2) If current variable is a multiple of 4 and is greater than 4: 8,12,14, ...
 - (a) Copy the 6th previous theta_v variable and add “pi”

- (b) e.g: θ_{vat} face number 12 = θ_v at face number 6 + π
- (c) If not a multiple of 4, the pull values from the iteration results: x

13) **Findingtheta**: calculating the joint angles for each joint for every cell in matrix

CC_CONVERT_SCFIX

- 1) Find all the fixed zvalues in xcell by using xcellvar
- 2) Initialize an emptied Φ_n matrix
- 3) Φ_n : Find ϕ_n by calculating the angles of deformation between any two fixed zvalues.
 - a) Only place the values that are not duplicates into Φ_n matrix
 - b) *Note: only the ϕ_n values will be changed for interpolating and initial guess
 - c) **Note: do not include the 4th n vectors on the fourth face
- 4) θ_{na} : Initialize the matrix of θ_n
- 5) θ_{va} : Initialize the matrix of θ_v
- 6) θ_n : condense by eliminating duplications
 - a) *Note: do not include the 4th n vectors on the fourth face
- 7) θ_v : condense by eliminating duplications
 - a) *Note: do not include the 4th v vectors on the fourth face

SC_GRAPHFIX

- 1) Uncondensing the variables: ϕ_n , θ_n , θ_v as shown in **realfree**
- 2) Set flagplot=1
- 3) **image3dfix**: plots the 3d images of the unit cell
- 4) Move the all the unit cells back to original starting locations before being shifted to origin
- 5) **graph_ctpt**: graphs the centerpoints

MINIMIZEPENERGYFIX

- 1) Expands the condensed matrices of S.C. coordinates vectors
- 2) Creates relationship among vectors based upon duplications
 - a) Skip the 4th v and n vectors on the 4th face of the 1st unit cell
- 3) Uncondensing the variables: phi_n, theta_n, theta_v as seen in **fixface**
- 4) Reminder:
 - a) Numbering conventions for the v vectors and n vectors on each unit cells

3	7	11
4 + 2	8 + 6	12 + 10
1	5	9

- b) Numbering in reference in face number For all the sections in each cell in a matrix is m-by-n matrix
 - i) One unit cell has 4 sections.
 - ii) phi_n
 - iii) if current variable is a multiple of 4 and is greater than 4: 8,12,14,...
 - (1) Copy the 6th previous phi_n variables
 - (2) e.g: phi_n at face number 12 = phi_n at face number 6
 - iv) If not a multiple of 4, the pull values from the iteration results: x
 - v) Theta_n
 - vi) if current variable is a multiple of 4 and is greater than 4: 8,12,14,...
 - (1) Copy the 6th previous theta_n variables
 - (2) e.g: theta_n at face number 12 = theta_n at face number 6
 - vii) If not a multiple of 4, the pull values from the iteration results: x
 - viii) Theta_v
 - ix) if current variable is a multiple of 4 and is greater than 4: 8,12,14,...
 - (1) Copy the 6th previous theta_v variable and add “pi”
 - (2) e.g: theta_v at face number 12 = theta_v at face number 6 + pi
 - x) If not a multiple of 4, the pull values from the iteration results: x
- 5) **image3dfix**: function that calculates the joint angles for minimizing
- 6) Can also calculate the center-points of each unit cell for plotting
- 7) Pot_energy: calculates the energy in the system from the angles of the joints

IMAGE3DFIX

- 1) Converting the N and V vectors from Spherical to cartesian for calculations
- 2) Finding vertices
 - a. Calculate the side a unit vectors
 - b. Calculate the middle a unit vector
 - c. Multiply the a unit vectors by their geometric design length
 - d. Find the Cartesian coordinates of the corners and centerpoints of every unit cell
 - e. vert_cn: Initialize a matrix for the centerpoints
 - f. Calculate Cartesian coordinates by multiplying every v_vectors by their corresponding geometric design length.
- 3) Graph by organizing the scalar a and v vectors by groups of three's for creating a 3d graphs using the embed MATLAB function "patching"
- 4) Change all vectors back to unit vectors.
- 5) Calculate the u unit vectors
- 6) Calculate the theta's: joint angles

FINDINGTHETA

- 1) Converting the N and V vectors from Spherical to cartesian for calculations
- 2) Cal joint angles
 - a) Calculate the side a unit vectors
 - b) Calculate the middle a unit vector
 - c) Multiply the a unit vectors by their geometric design length
 - d) Find the Cartesian coordinates of the corners and centerpoints of every unit cell
 - e) vert_cn: Initialize a matrix for the centerpoints
 - f) Calculate Cartesian coordinates by multiplying every v_vectors by their corresponding geometric design length.
 - g) Change all vectors back to unit vectors.
 - h) Calculate the u unit vectors
 - i) Calculate the theta's: joint angles

FIXFACE

```
close all;
clear all;

global k1;
global k2;
global k_alpha;
global k_beta;
global k_lambda;
global h_scaling;
global phi;
global a;
global v_vectors;
global n_norm;
global phi_ni;
global theta_ni;
global theta_vi;
global theta_tab;
global ncells;
global xcell;
global dist;
global xcellvar;

k1=500;
k2=800;
dist=.4905;
ncells=[1,3];
xcell=[0,    0, 0;
       0,  dist, 0;
       0, dist*2, 0];

k_alpha= (58.054*pi)/180;
k_beta=(22*pi)/180;
h_scaling=1/(cos((k_beta)/2));
%w_norm=tan((beta)/2);

%disp= 0.01; % inches
%r_arm=0.25;

%%%%%%%%%creating the 'on' 'off' matrix for what value canchange%%%%%%%%%
xcellvar=zeros(size(xcell));
```

```

counter31=0;
for it30= 1:ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1;
        if xcell(counter31,3)==0 %if xcell=0 note: '0.01' is not '0'
            xcellvar(counter31,1)=1;%then put these 1 or 0 into xcellvar
            xcellvar(counter31,2)=1;%1=free
            xcellvar(counter31,3)=1;
        else
            xcellvar(counter31,1)=1;
            xcellvar(counter31,2)=1;
            xcellvar(counter31,3)=0; %0=fixed
        end
    end
end
xcellvar;

%%%%%%%%%%%%additional constraints %%%%%%%%%%%%%%
%
% LT%    upper Lt 0 0 0 0 upperRT
% S%      0 0 0 0 0 0
% I%      0 0 0 0 0 0      ^
% D%      1stPT 0 0 0 0 LowerRT | Y
% E
%      -----Bottom row-----
%      -----X->-----> X

xcellvar(1,:)=[0 0 0];
% nXm size of the xcell_bdpt
%constraint the first pt with all '0' b/c 1st pt is constraint

%%%Constraining the various coordinate at the corners of the matrix %%%

xcellvar(ncells(2)*ncells(1)-ncells(2)+1,3)=0;
%constraint the upper LT Z coodinate
xcellvar(ncells(2)*ncells(1)-ncells(2)+1,1)=0;
%constraint the upper LT x coodinate
xcellvar(ncells(2),3)=0; %constraint the lower RT Z coodinate
xcellvar(ncells(2),2)=0; %constraint the lower RT 1 coodinate

if ncells(1)==1
    xcellvar(:,2)=0;
end

if ncells(2)==1

```

```

        xcellvar(:,1)=0;
    end

    xcellvar(ncells(1)*ncells(2),3)=0;
        %constraint the upper RT Z coodinate

    [Lx,Ly] =link_length (ncells, xcell);
        %function that finds the length of link:...
        %this will show what the intial guess gives

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% initial guess %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    [phi_n,theta_n,theta_v]= cc_convert_scfix(xcell,xcellvar);
    %interpolation function

    phi_ni=phi_n
    theta_ni=theta_n
    theta_vi=theta_v

    x0=[phi_n; %should be condensed
        theta_n;
        theta_v];

    x=x0;
    x
    [xcell]=sc_graphfix(x); %function that graphs

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% bounds for options %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %lower bound for searching for guess
    lower_b=[];
    for it=1:3*ncells(1)*ncells(2);
        lower_bi=-60*pi/180;  %phi_ni's

        lower_b=[lower_b;lower_bi];
    end

    for it=1:3*ncells(1)*ncells(2);

        lower_bi=-60*pi/180; %theta_ni
        lower_b=[lower_b;lower_bi];
    end

```

```

end

for it=1:3*ncells(1)*ncells(2);

    lower_bi=theta_vi(it)-60*pi/180; %theta_vi
    lower_b=[lower_b;lower_bi];

end

%upper bound for searching for guess
upper_b=[];
for it=1:3*ncells(1)*ncells(2);

    upper_bi=60*pi/180; %phi_ni's
    upper_b=[upper_b;upper_bi];

end

for it=1:3*ncells(1)*ncells(2);

    upper_bi=60*pi/180; %theta_ni
    upper_b=[upper_b;upper_bi];

end

for it=1:3*ncells(1)*ncells(2);

    upper_bi=theta_vi(it)+60*pi/180; %theta_vi
    upper_b=[upper_b;upper_bi];

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
tic;
options=optimset('Display','iter','MaxIter',1e5, 'MaxFunEvals',1e12,'TolFun',0.001,
'TolX',0.001);
%x = fmincon('energy_min',x0)

A_matrix=[];

x = fmincon('minimizePenergyfix',x0,[],[],[],[],lower_b, upper_b,[],options);

toc;
sec_elapsed_fvd_theta=toc

```

```
[xcell]=sc_graphfix(x); %function that graphs
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%  
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
cnt=0;  
phi_n=[];  
for it=1:4*ncells(1)*ncells(2)  
    cnt=cnt+1;  
    if it==4  
        phi_n1=0;  
        phi_n=[phi_n;phi_n1];  
        cnt=cnt-1;  
        % resetting counter to previous cnt becaused skipped a number  
    elseif mod(it,4)==0 %MOD Modulus (signed remainder after division).  
        phi_n3=phi_n(it-6);  
        phi_n=[phi_n;phi_n3];  
        cnt=cnt-1;% resetting counter to previous cnt becaused skipped a number  
    else  
        phi_n2=x(cnt);  
        phi_n=[phi_n;phi_n2];  
  
    end  
end
```

```
theta_n=[];  
for it=1:4*ncells(1)*ncells(2)  
    cnt=cnt+1;  
    if it==4  
        theta_n1=0;  
        theta_n=[theta_n; theta_n1];  
        cnt=cnt-1; % resetting counter to previous cnt becaused skipped a number  
    elseif mod(it,4)==0 %MOD Modulus (signed remainder after division).  
        theta_n3=theta_n(it-6);  
        theta_n=[theta_n;theta_n3];  
        cnt=cnt-1; % resetting counter to previous cnt becaused skipped a number  
    else  
        theta_n2=x(cnt);  
        theta_n=[theta_n; theta_n2];  
  
    end  
end
```

```
theta_v=[];  
for it=1:4*ncells(1)*ncells(2)  
    cnt=cnt+1;
```

```

if it==4
    theta_v1=pi;
    theta_v=[theta_v;theta_v1];
    cnt=cnt-1; % resetting counter to previous cnt becaused skipped a number
elseif mod(it,4)==0 %MOD Modulus (signed remainder after division).
    theta_v3=theta_v(it-6)+pi;
    theta_v=[theta_v;theta_v3];
    cnt=cnt-1; % resetting counter to previous cnt becaused skipped a number
else
    theta_v2=x(cnt);
    theta_v=[theta_v;theta_v2];
end
end

```

```

flagplot=0; % 0 for not plotting graphs
[theta_tab, vert_centpt] = image3dfix(phi_n, theta_n, theta_v,flagplot);
    % Kinematic functions that produces the thetas fo minmizations

```

```

theta_tab

```

CC_CONVERT_SCFIX

```

function [phi_n,theta_n,theta_v]= cc_convert_scfix(xcell,xcellvar);
%%from cartesian coordinates to spherical

global k1;
global k2;
global k_alpha;
global k_beta;
global k_lambda;
global h_scaling;
global phi;
global a;
global v_vectors;
global n_norm;
global phi_ni;
global theta_ni;
global theta_vi;
global theta_tab;
global ncells;
global xcell;
global dist;
global xcellvar;

%%%%%%%%step 1:=%%%%%%%% step 1:= %%%%%%%%%%
phi_n=[];
if ncells(2)>1;
    %find the index of all the centerpts in the 1st row that Z is inputed
    Vrow= find(xcellvar(1:ncells(2),3)==0)
    for it=2:size(Vrow)
        phi_n_interp1=-asin((xcell(Vrow(it),3)-xcell(Vrow(it-1),3))...
            /((Vrow(it)-Vrow(it-1))*dist)); %finding the angles of deformation
        if it~=2 %not
            phi_n(length(phi_n)-1)=phi_n_interp1; %n6
            phi_n(length(phi_n))=(phi_n_interp2+phi_n_interp1)/2; %n7
            phi_n(length(phi_n)-2)=(phi_n_interp2+phi_n_interp1)/2; %n5
        end
        if it==2
            for it2=1:4*((Vrow(it)-Vrow(it-1))+1);
                if mod(it2,4)~=0 %skipping the n4, n8....
                    %phi_n1=phi_n_interp1*ones((vrowl(it)-vrowl(it-1))*4,1);
                    phi_n=[phi_n; phi_n_interp1];
                end
            end
        end
    end
end

```

```

        end

    end
else
    for it2=1:4*((Vrow(it)-Vrow(it-1)));
        if mod(it2,4)~=0 %skipping the n4, n8....
            %phi_n1=phi_n_interp1*ones((vrow1(it)-vrow1(it-1))*4,1);
            phi_n=[phi_n; phi_n_interp1];

        end

    end

end

    end
    phi_n_interp2=phi_n_interp1;
end
end

phi_n_interp1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%% spherical: n01=[ phi, theta] because r is always 1
%%%%%%%% theta= rotating latitude
%%%%%%%% phi= rotatin longitude

%phi_n(4)=0; %Because the first pt=0,0,0, There n4 remains fixed

theta_na=zeros(4*ncells(1)*ncells(2),1);

theta_va=[];
for it=1:4*ncells(1)*ncells(2);
    theta_va(it,1)=3*pi/2;
    theta_va(it+1,1)=0;
    theta_va(it+2,1)=pi/2;
    theta_va(it+3,1)=pi;
end
theta_n=[];
for it=1:4*ncells(1)*ncells(2);
    if mod(it,4)~=0 %to condense by eliminating duplicates
        theta_ni=theta_na(it,1);
        theta_n=[theta_n;theta_ni];
    end
end

theta_v=[ ];

```



```
for it=1:4*ncells(1)*ncells(2);  
    if mod(it,4)~=0  
        theta_vi=theta_va(it,1);  
        theta_v=[theta_v;theta_vi];  
    end  
end
```

```
phi_ni=phi_n;  
theta_ni=theta_n;  
theta_vi=theta_v;
```

SC_GRAPHFIX

```
function [xcell]=sc_graphfix(x)
%graphing the spherical values by uncompressing and changing to cartesians

global k1;
global k2;
global k_alpha;
global k_beta;
global k_lambda;
global h_scaling;
global phi;
global a;
global v_vectors;
global n_norm;
global phi_ni;
global theta_ni;
global theta_vi;
global theta_tab;
global ncells;
global xcell;
global dist;
global xcellvar;

%%%%%%%% %%%%%%%%%%uncompression.....%%%%%%%% %%%%%%%%%% %%%%%%%%%%
%Has the repeats where all the n's that are parellel will be listed as well
%%%%%%%% %%%%%%%%%% At the begining all Phi_ni=Phi_n,theta_ni=theta_n, and theta_vi=theta_v
%%%%%%%% %%%%%%%%%% THe Phi_ni, etc were uses as initial guesses, while the Phi_n, etc.
%%%%%%%% %%%%%%%%%% were kept as the orginal inputs and deformation
%%%%%%%% %%%%%%%%%% In this section replace the results X values from 'fmincon' into
%%%%%%%% %%%%%%%%%% the Phi_n,etc. but do not replace the constrained values.

cnt=0;
phi_n=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;
    if it==4
        phi_n1=0;
        phi_n=[phi_n;phi_n1];
        cnt=cnt-1;% reseting counter to previous cnt becaused skipped a number
    elseif mod(it,4)==0 %MOD    Modulus (signed remainder after division).
        phi_n3=phi_n(it-6);
        phi_n=[phi_n;phi_n3];
        cnt=cnt-1;% reseting counter to previous cnt becaused skipped a number
```

```

else
    phi_n2=x(cnt);
    phi_n=[phi_n;phi_n2];

end
end

theta_n=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;
    if it==4
        theta_n1=0;
        theta_n=[theta_n; theta_n1];
        cnt=cnt-1; % resetting counter to previous cnt becaused skipped a number
    elseif mod(it,4)==0 %MOD Modulus (signed remainder after division).
        theta_n3=theta_n(it-6);
        theta_n=[theta_n;theta_n3];
        cnt=cnt-1; % resetting counter to previous cnt becaused skipped a number
    else
        theta_n2=x(cnt);
        theta_n=[theta_n; theta_n2];
    end
end
end

```

```

theta_v=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;
    if it==4
        theta_v1=pi;
        theta_v=[theta_v;theta_v1];
        cnt=cnt-1; % resetting counter to previous cnt becaused skipped a number
    elseif mod(it,4)==0 %MOD Modulus (signed remainder after division).
        theta_v3=theta_v(it-6)+pi;
        theta_v=[theta_v;theta_v3];
        cnt=cnt-1; % resetting counter to previous cnt becaused skipped a number
    else
        theta_v2=x(cnt);
        theta_v=[theta_v;theta_v2];
    end
end
end

```

```

%%%%%%%%%%
%%%%%%%%%%
%phi_n=[x(1); phi_ni(2); x(2); phi_ni(4)];
%theta_n=[x(3); theta_ni(2); x(4); theta_ni(4)];
%theta_v=[x(5); theta_vi(2); x(6); theta_vi(4)];

```

```

flagplot=1; % for plotting graphs in fvd_patching

[theta_tab, vert_centpt] = image3dfix(phi_n, theta_n, theta_v, flagplot);
% Kinematic program that produces the thetas fo minmizations

xcell=vert_centpt;
[Lx, Ly] =link_length (ncells, xcell);
%function that finds the length of link: this will show what the intial guess gives
graph_ctpt (ncells, xcell);% a function that graphs the center pt

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

MINIMIZINGPENERGYFIX

```
function [pot_energy] = minimizePenergyfix(x)

global k1;
global k2;
global k_alpha;
global k_beta;
global k_lambda;
global h_scaling;
global phi;
global a;
global v_vectors;
global n_norm;
global phi_ni;
global theta_ni;
global theta_vi;
global theta_tab;
global ncells;
global xcell;
global dist;
global xcellvar;

%%%%% spherical: n01=[ phi, theta] because r is always 1
%%%%% theta= rotating latitude
%%%%% phi= rotatin longitude
%%%%%%%%%Creating relations between vectors%%%%%%%%%
%%This code sets any normals that should be parellel is parellet.%% %%%%%%%%%%

%example: n2= n8 because both vectors are on the linking links

cnt=0;
phi_n=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;
    if it==4
        phi_n1=0;
        phi_n=[phi_n;phi_n1];
        cnt=cnt-1;% reseting counter to previous cnt becaused skipped a number
    elseif mod(it,4)==0 %MOD Modulus (signed remainder after division).
        phi_n3=phi_n(it-6);
        phi_n=[phi_n;phi_n3];
        cnt=cnt-1;% reseting counter to previous cnt becaused skipped a number
    else
```

```

    phi_n2=x(cnt);
    phi_n=[phi_n;phi_n2];

end
end

%theta_n=theta_ni;
theta_n=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;
    if it==4
        theta_n1=0;
        theta_n=[theta_n; theta_n1];
        cnt=cnt-1; % resetting counter to previous cnt because skipped a number
    elseif mod(it,4)==0 %MOD Modulus (signed remainder after division).
        theta_n3=theta_n(it-6);
        theta_n=[theta_n;theta_n3];
        cnt=cnt-1; % resetting counter to previous cnt because skipped a number
    else
        theta_n2=x(cnt);
        theta_n=[theta_n; theta_n2];
    end
end

theta_v=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;
    if it==4
        theta_v1=pi;
        theta_v=[theta_v;theta_v1];
        cnt=cnt-1; % resetting counter to previous cnt because skipped a number
    elseif mod(it,4)==0 %MOD Modulus (signed remainder after division).
        theta_v3=theta_v(it-6)+pi;
        theta_v=[theta_v;theta_v3];
        cnt=cnt-1; % resetting counter to previous cnt because skipped a number
    else
        theta_v2=x(cnt);
        theta_v=[theta_v;theta_v2];
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

flagplot =0; % for not plotting graphs
[theta_tab, vert_centpt] = image3dfix(phi_n, theta_n, theta_v,flagplot);

```

```

% Kinematic functions that produces the thetas fo minmizations

theta_tab;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5

pot_energy=0;

for it2=1:ncells(1)*ncells(2)
    for it=1:3:3*4; %for every sections there are three angles

        pot_energy1=1/2 *k1*(115.1005*pi/180-theta_tab(it,it2) )^2 ...
            +1/2 *k2*(82.6897*pi/180-theta_tab(it+1,it2))^2 ...
            +1/2 *k1*(115.1005*pi/180-theta_tab(it+2,it2))^2;

        pot_energy=pot_energy+pot_energy1;
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
creating the 'on/' 'off' matrix for what value can change%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

counter31=0;% interiopr cell conunter
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1; %%% interiopr cell conunter Start at the 1st interior cell

        if xcell(counter31,3) ~=0 %if xcell=0 note: '0.0000001' is not '0'
            pot_energy=pot_energy+...
                100000*(vert_centpt(counter31,3)-xcell(counter31,3))^2;
        end
    end
    %counter31=counter31+2;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

close
%convert back to spherical for the 'fvd_constraint' function

```

IMAGE3DFIX

```
function [theta_tab, vert_centpt] = image3dfix(phi_n, theta_n, theta_v, flagplot);
% Kinematic program that produces the thetas for minimizations

global k1;
global k2;
global k_alpha;
global k_beta;
global k_lambda;
global h_scaling;
global phi;
global a;
global v_vectors;
global n_norm;
global phi_ni;
global theta_ni;
global theta_vi;
global theta_tab;
global ncells;
global xcell;
global dist;
global xcellvar;

%%%%%%%% spherical: n01=[ phi, theta] because r is always 1
%%%%%%%% theta= rotating latitude
%%%%%%%% phi= rotating longitude
%%%%%%%% There is a relationship of the phi for v vectors and the n vector locations

%converting the N and V vectors back to cartesian/rectangular for calculations%%%%%%%%

n_norm=[];
%%converting back to rectangular for calculations
for it=1:4*ncells(1)*ncells(2)
    n_norm(it,:)=sin(phi_n(it))*cos(theta_n(it)),...
                sin(phi_n(it))*sin(theta_n(it)), cos(phi_n(it));
end

v_vectors=[];
%converting back to rectangular for calculations
for it= 1:4*ncells(1)*ncells(2)
    v_vectors(it,:)=cos(phi_n(it))*cos(theta_n(it))*cos(theta_v(it))...
                    - sin(theta_n(it))*sin(theta_v(it)), ...
                    cos(phi_n(it))*sin(theta_n(it))*cos(theta_v(it))...
                    + cos(theta_n(it))*sin(theta_v(it)), ...
```



```

        -sin(phi_n(it))*cos(theta_v(it)));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Calculating Angles %%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

a=[]; %a=side vector of top small connecting triangles

for it21=1:4*ncells(1)*ncells(2)
    a_val1=([v_vectors(it21,:)]-tan(k_beta/2)*(cross(n_norm(it21,:),...
        v_vectors(it21,:)))/(h_scaling); %vp1 +w_rot1
    a=[a;a_val1];
    a_val2=([v_vectors(it21,:)]+tan(k_beta/2)*(cross(n_norm(it21,:),...
        v_vectors(it21,:)))/(h_scaling); %vp1 +w_rot2
    a=[a;a_val2];
end

a; %%% a8 %a1 %a2 %a3.....
%a=example for 1 unit cell:[a8; a1; a2; a3; a4; a5; a6; a7]
%rearrange the a matrix to move the first a value from
%each unit cell become the last value for each unit cell
a2=[]; %a1 %a2 %a3.....%a8

for it23= 1:8*ncells(1)*ncells(2)-7
    a1=[a(it23+1:it23+7,:);a(it23,:)];
    a2=[a2;a1];
end
a=a2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

m=[];
;
count4=0;
%for changing the vector 1 and vector2 from the 'a' matrix for 'For Loop it6'
%finding a2z=az_mid1 by searching through trial and error: in 'For Loop it5'
L1= 73/256; %%length of the side of intermediate joints
L5=83/256; %%length of the middle vector of intermediate joints
for it6=1:4*ncells(1)*ncells(2) % the total numbers of a_mid for all cells
    count4=count4+1;
    vect1=a(count4,:);
    count4=count4+1;

```

```

vect2=a(count4,:);

p_vect=0.25*vect2 - 0.25*vect1;
    %%% p_vect= a vector that connects a(i) and a(i+1).
q_vect=0.25*vect1 + p_vect/2;
    %%%q_vect= vector taht connects the centerpt of unit cell to mid of p_vect.
L4= sqrt(L1^2 - (norm(p_vect/2))^2);
    %% length that connects Q_vector to the bottom of the triangle
angle_triangle= acos( -(L4^2)+(norm(q_vect))^2 +L5^2)/(2*norm(q_vect)*L5));
L6= norm(q_vect)*tan(angle_triangle);
    %%L6= perpendicular from end of q_vector to L5
n_avects= cross(vect1,vect2);
    %%% normal vector to 2 crossing a vectors
n_avectu= n_avects/(norm(n_avects)); %%% the unit vector of the a vector normal
m_vect= -L6*n_avectu+q_vect;
m_vectu= m_vect/(norm(m_vect));
m=[m;m_vectu];
end

ap=a*0.25; %scaler of a vectors
a_mid= m*(83/256) ; %scaler of m_mid vectors

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%turning the C.C of V and N into the C.C. of the centerpts for each unit cell%%

%pat_vert=[0,0,0];
%vert_cn=[0,0,0];

vert_cn=zeros(ncells(1)*ncells(2),3);
%starting off the vertices of the centerpts for each unit cell

%cnt=0;
%for it=2:4:4*ncells(1)*ncells(2)-4
% cnt=cnt+1;
% vert_cn1=vert_cn(cnt,:)+v_vectors(it,:)*dist;
% vert_cn=[vert_cn; vert_cn1];
%end

% calculates all the vertrices pts
cnt=0;
pat_vert=[];
cnt2=0;
for it2= 1:ncells(1)*ncells(2) % going through each unit cell
    pat_vert=[pat_vert; vert_cn(it2,:)];
    %first intializes the first vertices which is the centerpt for each unit cell

```

```

for it=1:4
    cnt=cnt+1;
    cnt2=cnt2+1;

    % adds in the vertices for the scalar "a vectors" and "middle vectors"
    pat_vert1=[ap(cnt,:);
        a_mid(cnt2,:)];
    pat_vert=[pat_vert; pat_vert1]; %stacking
    cnt=cnt+1;
    pat_vert2=ap(cnt,:);
    pat_vert=[pat_vert; pat_vert2];
end
end
pat_vert;
%translating groups of 13points all at the same time to the scalar postion
%for each unit cell
cnt=0;
cnt2=13; %starting at the 13th vertices which is on the 2nd cell
for it=2:ncells(1)*ncells(2)
    for it2=1:13 %for every cell there is 13 vertice points
        cnt2=cnt2+1; %conuter to count for each vertices
        pat_vert(cnt2,:)=pat_vert(cnt2,:)+pat_vert(13*cnt+1,:)+v_vectors(2+cnt*4,:)*dist;
    end
    cnt=cnt+1;
end

pat_vert;

cfaces=[];
vert_centpt=[];

cnt=0;
for it=1:ncells(1)*ncells(2)
    for it2=1:11
        cfaces1=[13*cnt+it2+1,13*cnt+1,13*cnt+it2+2];

        cfaces=[cfaces; cfaces1];
    end
    vert_centpt1=pat_vert(13*cnt+1,:);
    %placing all of the center points of each unit cell into one matrix
    vert_centpt=[vert_centpt;vert_centpt1];
    cfaces2=[13*cnt+2 ,13*cnt+1,13*cnt+13];
    cfaces=[cfaces;cfaces2];
    cnt=cnt+1;
end
vert_centpt;

```

```

cfaces;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%cfaces=[2,1,3; %1
%      3,1,4; %2
%      4,1,5;
%      5,1,6; %3
%      6,1,7; %4
%      7,1,8;
%      8,1,9; %5
%      9,1,10;
%      10,1,11;%6
%      11,1,12;%7
%      12,1,13;%8
%      2,1,13]; %12
if flagplot==1;
color_vect=[]; %for RGB

for it=1:ncells(1)*ncells(2)
    for it2=1:4
        color_vect1=[0,0,1; %blue face in RGB
        0,1,0; %green face in RGB
        1,0,0]; %red face in RGB
        color_vect=[color_vect; color_vect1];
    end
end

patch('Vertices',pat_vert,'Faces',cfaces,'FaceVertexCData',color_vect,'FaceColor','flat')
view(3);
axis equal;
pause(.2)
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%change back to unit vectors

m_unit=[];
for it=1:4*ncells(1)*ncells(2)
    m1= a_mid(it,:)/(norm(a_mid(it,:)));
    m_unit=[m_unit;m1];
end
m=m_unit;

```

```

a_unit=[];
for it =1:8*ncells(1)*ncells(2)
    a1= ap(it,:)/(norm(ap(it,:)));
    a_unit=[a_unit;a1];

end

ap=a_unit;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%findin the U vectors that are normal to the intermediate links using the 'a' and 'a_mid'
vectors
%a=example for 1 unit cell:[ a1; a2; a3; a4; a5; a6; a7; a8]
%a_mid= [a_mid1(between a1 and a2); a_mid2(between a3 and a4); a_mid3(between a5
and a6); a_mid4(between a7 and a8)]

u_norm=[];
count5=0;
for it7=1:4*ncells(1)*ncells(2)
    count5=count5+1;
    u_val1=( cross(ap(count5,:),a_mid(it7,:)) )/...
        ( norm (cross(ap(count5,:),a_mid(it7,:)) ));
    u_norm=[u_norm;u_val1];

    count5=count5+1;
    u_val2= ( cross(a_mid(it7,:), ap(count5,:)) )/...
        ( norm (cross(a_mid(it7,:), ap(count5,:)) ));
    u_norm=[u_norm;u_val2];
end
u_norm; %example for 1 unit cell:[8 rows, 3 colns] [ u1; u2; u3; u4; u5; u6; u7; u8]

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% explanation of finding theta
% find theta1 (between main and intermediate), theta2 (between 2 intermediate), theta3
(between main and intermediate)
% g_n= [g_n1; g_n2; g_n3; g_n4];

% from paul's eqns:
% theta1=- acos(dot(u1,n1)+pi
% theta3= -acos(dot(u2,n2)+pi    modified from Paul's eqn: '+' => '-'.
% theta 2= -acos(u1,u2)+ pi

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

theta=[];
% for every 3 rows are the 3 thetas for 1 out of 4 section...
%of the radial symetric unit cell.

```

```

%copying the beginning the 1st norm to the end for: 'For loop it8' because of circular
motion
g_n_it8= [n_norm(1:4*ncells(1)*ncells(2),:); n_norm(1,:)];
count6=0;
for it8= 1:4*ncells(1)*ncells(2)
    count6=count6+1;
    theta1= -acos( dot(u_norm(count6,:), g_n_it8(it8,:)) ) + pi ;
    theta=[theta;theta1];

    theta2= -acos( dot(u_norm(count6,:), u_norm(count6+1,:)) ) +pi;
    theta=[theta;theta2];

    theta3= -acos( dot(u_norm(count6+1,:), g_n_it8(it8+1,:)) ) + pi ;
    theta=[theta;theta3];
    count6=count6+1;

end
theta;

theta_tab=[];
for it24=1:12:3*4*ncells(1)*ncells(2)
    theta_t=[theta(it24:it24+11)];
    theta_tab=[theta_tab,theta_t];
end

theta_tab;

```

FINDINGTHETA

```
function [theta_tab, vert_centpt] = findingtheta(phi_n, theta_n, theta_v); % Kinematic
%program that produces the thetas fo minmizations
```

```
global k1;
global k2;
global k_alpha;
global k_beta;
global k_lambda;
global h_scaling;
global phi;
global a;
global v_vectors;
global n_norm;
global phi_ni;
global theta_ni;
global theta_vi;
global theta_tab;
global ncells;
global xcell;
global dist;
global xcellvar;
global shift_xcell;
```

```
%%%%%%%% spherical: n01=[ phi, theta] because r is always 1
```

```
%%%%%%%% theta= rotating latitude
```

```
%%%%%%%% phi= rotatin longitude
```

```
%%%%%%%%There is a relationship of the phi for v vectors and the n vector locations
```

```
%%%%%%%%%converting the N and V vectors back to cartesian for calculations%%%%%%%%%
```

```
n_norm=[];
```

```
%%converting back to rectangular for calculations
```

```
for it=1:4*ncells(1)*ncells(2)
```

```
    n_norm(it,:)=sin(phi_n(it))*cos(theta_n(it)),...
```

```
    sin(phi_n(it))*sin(theta_n(it)), cos(phi_n(it));
```

```
end
```

```
v_vectors=[];
```

```
%converting back to rectangular for calculations
```

```
for it= 1:4*ncells(1)*ncells(2)
```

```
    v_vectors(it,:)=cos(phi_n(it))*cos(theta_n(it))*cos(theta_v(it)) ...
```

```
        - sin(theta_n(it))*sin(theta_v(it)), ...
```

```
        cos(phi_n(it))*sin(theta_n(it))*cos(theta_v(it))...
```

```
        + cos(theta_n(it))*sin(theta_v(it)), ...
```

```

        -sin(phi_n(it))*cos(theta_v(it))];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%calculating angles%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

a=[]; %a=side vector of top small connecting triangles

for it21=1:4*ncells(1)*ncells(2)
    a_val1=([v_vectors(it21,:)]-tan(k_beta/2)*(cross(n_norm(it21,:),...
        v_vectors(it21,:)))/(h_scaling); %vp1 +w_rot1
    a=[a;a_val1];
    a_val2=([v_vectors(it21,:)]+tan(k_beta/2)*(cross(n_norm(it21,:),...
        v_vectors(it21,:)))/(h_scaling); %vp1 +w_rot2
    a=[a;a_val2];
end

a; %%%% a8 %a1 %a2 %a3.....
%a=example for 1 unit cell:[a8; a1; a2; a3; a4; a5; a6; a7]
%rearrange the a matrix to move the first a value from each...
%unit cell become the last value for each unit cell
a2=[]; %a1 %a2 %a3.....%a8

for it23= 1:8:8*ncells(1)*ncells(2)-7
    a1=[a(it23+1:it23+7,:);a(it23,:)];
    a2=[a2;a1];
end
a=a2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
m=[];
;
count4=0;
%for changing the vector 1 and vector2 from the 'a' matrix for 'For Loop it6'
%finding a2z=az_mid1 by searching through trial and error: in 'For Loop it5'
L1= 73/256; %%%length of the side of intermediate joints
L5=83/256; %%%length of the middle vector of intermediate joints
for it6=1:4*ncells(1)*ncells(2) % the total numbers of a_mid for all cells
    count4=count4+1;
    vect1=a(count4,:);
    count4=count4+1;
    vect2=a(count4,:);

    p_vect=0.25*vect2 - 0.25*vect1; %%%% vector that connects a(i) and a(i+1).
    q_vect=0.25*vect1 + p_vect/2;
    %%%% vector taht connects the centerpt of unit cell to mid of p_vect.
    L4= sqrt(L1^2 - (norm(p_vect/2))^2);

```



```

        %% length that connects Q_vector to the bottom of the triangle
angle_triangle= acos( -(L4^2)+(norm(q_vect))^2 +L5^2)/(2*norm(q_vect)*L5));
L6= norm(q_vect)*tan(angle_triangle);
        %%L6= perpendicular from end of q_vector to L5
n_avects= cross(vect1,vect2); % normal vector to 2 crossing a vectors
n_avectu= n_avects/(norm(n_avects)); % the unit vector of the a vector normal
m_vect= -L6*n_avectu+q_vect;
m_vectu= m_vect/(norm(m_vect));
m=[m;m_vectu];
end

ap=a*0.25; %scaler of a vectors
a_mid= m*(83/256) ; %scaler of m_mid vectors

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%turning the C.C of V and N into the C.C. of the centerpts for each unit cell%%

pat_vert=[0,0,0];
vert_cn=[0,0,0];

vert_cn=zeros(ncells(1)*ncells(2),3);
        %starting off the vertices of the centerpts for each unit cell

% calculates all the vertrices pts
cnt=0;
pat_vert=[];
cnt2=0;
for it2= 1:ncells(1)*ncells(2) % going through each unit cell
    pat_vert=[pat_vert; vert_cn(it2,:)];
    %first intializes the first vertices which is the ctpt for each cell
    for it=1:4
        cnt=cnt+1;
        cnt2=cnt2+1;
        pat_vert1=[ap(cnt,:);
            % adds in the vertices for the scaler "a vectors" and "middle vectors"
            a_mid(cnt2,:)];
        pat_vert=[pat_vert; pat_vert1]; %stacking
        cnt=cnt+1;
        pat_vert2=ap(cnt,:);
        pat_vert=[pat_vert; pat_vert2];
    end
end
pat_vert;

        %translating groups of 13points all at the same time to the scalar postion
        %for each unit cell

```

```

cnt=0;
cnt2=13; %starting at the 13th vertices which is on the 2nd cell
for it=2:ncells(1)*ncells(2)
    for it2=1:13 %for every cell there is 13 vertice points
        cnt2=cnt2+1; %counter to count for each vertices
        pat_vert(cnt2,:)=pat_vert(cnt2,:)+pat_vert(13*cnt+1,:)+v_vectors(2+cnt*4,:)*dist;
    end
    cnt=cnt+1;
end

pat_vert;

cfaces=[];
vert_centpt=[];

cnt=0;
for it=1:ncells(1)*ncells(2)
    for it2=1:11
        cfaces1=[13*cnt+it2+1,13*cnt+1,13*cnt+it2+2];

        cfaces=[cfaces; cfaces1];
    end
    vert_centpt1=pat_vert(13*cnt+1,:);
    %placing all of the ctps of each unit cell into one matrix
    vert_centpt=[vert_centpt;vert_centpt1];
    cfaces2=[13*cnt+2 ,13*cnt+1,13*cnt+13];
    cfaces=[cfaces;cfaces2];
    cnt=cnt+1;
end
vert_centpt;
cfaces;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%change back to unit vectors%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

m_unit=[];
for it =1:4*ncells(1)*ncells(2)
    m1= a_mid(it,:)/(norm(a_mid(it,:)));
    m_unit=[m_unit;m1];
end
m=m_unit;

a_unit=[];

```

```

for it=1:8*ncells(1)*ncells(2)
    a1= ap(it,:)/(norm(ap(it,:)));
    a_unit=[a_unit;a1];

end

ap=a_unit;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%findin the U vectors that are normal to the intermediate links using the 'a' and 'a_mid'
vectors
%a=example for 1 unit cell:[ a1; a2; a3; a4; a5; a6; a7; a8]
%a_mid= [a_mid1(between a1 and a2);...
%%%%%%%%%a_mid2(between a3 and a4);...
%%%%%%%%%a_mid3(between a5 and a6);...
%%%%%%%%%a_mid4(between a7 and a8)]

u_norm=[];
count5=0;
for it7=1:4*ncells(1)*ncells(2)
    count5=count5+1;
    u_val1=( cross(ap(count5,:),a_mid(it7,:)))/( norm (cross(ap(count5,:),a_mid(it7,:)) ));
    u_norm=[u_norm;u_val1];

    count5=count5+1;
    u_val2= ( cross(a_mid(it7,:), ap(count5,:)))/( norm (cross(a_mid(it7,:), ap(count5,:))
));
    u_norm=[u_norm;u_val2];
end
%example for 1 unit cell:[8 rows, 3 cols] [ u1; u2; u3; u4; u5; u6; u7; u8]

%%%%%%%%%explanation of finding theta%%%%%%%%%
% find theta1 (between main and intermediate),...
%theta2 (between 2 intermediate), theta3 (between main and intermediate)
% g_n= [g_n1; g_n2; g_n3; g_n4];

% from paul's eqns:
% theta1=- acos(dot(u1,n1))+pi
% theta3= -acos(dot(u2,n2))+pi   modified from Paul's eqn: '+' => '-'.
% theta 2= -acos(u1,u2)+ pi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```
theta=[]; % for every 3 rows are the 3 thetas for 1 out of 4 section of the radial symmetric
unit cell.
```

```
%copying the beginning the 1st norm to the end for: 'For loop it8' because of circular
motion
```

```
for it=0:ncells(1)*ncells(2)-1
    count6=0;
    for it8= 1:4
        count6=count6+1;
        theta1= -abs(acos( dot(u_norm(count6+it*8,:), n_norm(it8+it*4,:)))) + pi ;
        theta=[theta;theta1];

        theta2= -abs(acos( dot(u_norm(count6+it*8,:), u_norm((count6+1)+it*8,:)))) +pi;
        theta=[theta;theta2];

        if it8==4
            theta3= -abs(acos( dot(u_norm((count6+1)+it*8,:), n_norm(1+it*4,:)))) + pi ;
            theta=[theta;theta3];
            count6=count6+1;
        else
            theta3= -abs(acos( dot(u_norm((count6+1)+it*8,:), n_norm((it8+1)+it*4,:)))) + pi
;
            theta=[theta;theta3];
            count6=count6+1;
        end

    end
end

theta_tab=[];
for it24=1:12:3*4*ncells(1)*ncells(2)
    theta_t=[theta(it24:it24+11)];
    theta_tab=[theta_tab,theta_t];
end
```

APPENDIX E:

MATLAB FOR METHOD 2: FREE FACE

Method 2 without having any face being fixed was also implemented through using MATLAB Coding. The first page will be a guide to the code and the functions in the code. The rest of the pages in this Appendix are the code.

Navigation Outline

Program three implementing Method 2: Free face

Nomenclature:

k1= stiffness of type-1 joint
k2= stiffness of type-2 joint
k_alpha= geometric design constant
k_beta = geometric design constant
k_lambda= geometric design constant
h_scaling= geometric design constant
xcell: a matrix of the Cartesian coordinates of the center-points of the unit cells
ncells: the matrix dimension
xcellvar: controlling the constraints. Dimension size is the same as xcell.
dist: fixed distance value between any two point
xcell_initial: initial value of xcell before the modifying any values.
zinputs: user dislocation inputs for the z-values in xcell

REALFREE

- 1) specifying inputs
- 2) k1,k2,dist,ncells,xcell, k_alpha, k_beta, h_scaling
- 3) Constraining certain variables
- 4) Xcellvar: “on/off” matrix for controlling what value from xcell matrix can change
- 5) xcell: an (m*n)-by-3 matrix, where x is in the first column, y is 2nd, z is 3th
- 6) 0= off, meaning value should remain fix
- 7) 1=on, value can change
- 8) Shifting coordinates to origin
- 9) If the first unit cell has a z-displacement, z1, then shift all the coordinates in xcell z1 amount, until the first cell is back at the origin.
- 10) This will help simplify the calculation with the calculation starting at the origin.
- 11) Developing the initial guess

12) **cc_convert_sc:**

- a. converting the Cartesian coordinates to Spherical
- b. interpolating by rotating the S.C
- c. condensing the variables by realization of duplications

13) creating initial S.C. variables from previous function results:

- a. ϕ_{ni} , θ_{ni} , θ_{vi}

14) creating x0: initial guess vector

15) **sc_graph:** graphing the 3D matrix with the S.C. variables

16) Creating upper and lower bounds for x0

17) lower bound for ϕ_{ni} , θ_{ni} , θ_{vi}

18) upper bound for ϕ_{ni} , θ_{ni} , θ_{vi}

19) **Fmincon** –blackbox MATLAB function

- a. **minimizingPenergy:** calculating the potential energy in the system for Fmincon

20) Expands the condensed matrices of S.C. coordinates vectors

21) Creates relationship among vectors based upon duplications

22) Uncondensing the variables: ϕ_n , θ_n , θ_v

- a. Reminder:
- b. numbering conventions for the v vectors and n vectors on each unit cells
 - i. numbering in reference in face number

3	7	11
4 + 2	8 + 6	12 + 10
1	5	9

- c. For all the sections in each cell in a matrix is m-by-n matrix
- d. One unit cell has 4 sections.
 - i. ϕ_n
 - 1. if current variable is a multiple of 4 and is greater than 4: 8,12,14,...
 - 2. Copy the 6th previous ϕ_n variables
 - 3. e.g: ϕ_n at face number 12= ϕ_n at face number 6
 - 4. If not a multiple of 4, then pull values from the iteration results: x

- ii. θ_n

1. if current variable is a multiple of 4 and is greater than 4:
8,12,14,...
2. Copy the 6th previous theta_n variables
3. e.g: theta_n at face number 12 = theta_n at face number 6
4. If not a multiple of 4, then pull values from the iteration results: x

iii. Theta_v

1. if current variable is a multiple of 4 and is greater than 4:
8,12,14,...
2. Copy the 6th previous theta_v variable and add “pi”
3. e.g: theta_v at face number 12 = theta_v at face number 6
+pi
4. If not a multiple of 4, the pull values from the iteration results: x

23) **Findingtheta**: calculating the joint angles for each joint for every cell in matrix

CC_CONVERT_SC

- 1) Find all the fixed zvalues in xcell by using xcellvar
- 2) Initialize an emptied Phi_n matrix.
- 3) Phi_n: Find phi_n by calculating the angles of deformation between any two fixed zvalues.
 - a. Only place the values that are not duplicates into Phi_n matrix
 - b. *note: only the phi_n values will be changed for interpolating and initial guess
- 4) Theta_na: Initialize the matrix of theta_n
- 5) Theta_va: Initialize the matrix of theta_v
- 6) Theta_n: condense by eliminating duplications
- 7) Theta_v: condense by eliminating duplications

SC_GRAPH

- 1) Uncondensing the variables: phi_n, theta_n, theta_v as shown in **realfree**
- 2) Set flagplot=1
- 3) **image3d**: plots the 3d images of the unit cells
- 4) Move the all the unit cells back to original starting locations before being shifted to origin
- 5) **graph_ctpt**: graphs the centerpoints

MINIMIZEPENRGY

- 1) Expands the condensed matrices of S.C. coordinates vectors
- 2) Creates relationship among vectors based upon duplications
- 3) Uncondensing the variables: phi_n, theta_n, theta_v

4) Reminder:

- a. numbering conventions for the v vectors and n vectors on each unit cells
- b. numbering in reference in face number

3	7	11
4 + 2	8 + 6	12 + 10
1	5	9

5) For all the sections in each cell in a matrix is m-by-n matrix

- a. One unit cell has 4 sections.

i. ϕ_n

1. if current variable is a multiple of 4 and is greater than 4: 8,12,14,...
2. Copy the 6th previous ϕ_n variables
3. e.g: ϕ_n at face number 12 = ϕ_n at face number 6
4. If not a multiple of 4, the pull values from the iteration results: x

ii. θ_n

1. if current variable is a multiple of 4 and is greater than 4: 8,12,14,...
2. Copy the 6th previous θ_n variables
3. e.g: θ_n at face number 12 = θ_n at face number 6
4. If not a multiple of 4, the pull values from the iteration results: x

iii. θ_v

1. if current variable is a multiple of 4 and is greater than 4: 8,12,14,...
2. Copy the 6th previous θ_v variable and add “pi”
3. e.g: θ_v at face number 12 = θ_v at face number 6 + pi
4. If not a multiple of 4, the pull values from the iteration results: x

6) **image3d**: function that calculates the joint angles for minimizing

- a. Can also calculate the center-points of each unit cell for plotting

7) **Pot_energy**: calculates the energy in the system from the angles of the joints

IMAGE3D

- 1) Converting the N and V vectors from Spherical to cartesian for calculations
- 2) Calculate joint angles
 - a. Calculate the side a unit vectors
 - b. Calculate the middle a unit vector
 - c. Multiply the a unit vectors by their geometric design length
- 3) Find the Cartesian coordinates of the corners and centerpoints of every unit cell
- 4) vert_cn: Initialize a matrix for the centerpoints
- 5) Calculate Cartesian coordinates by multiplying every v_vectors by their
- 6) corresponding geometric design length.
- 7) Organize the scalar a and v vectors by groups of three's for creating a 3d graphs using the embed MATLAB function "patching".
- 8) Change all vectors back to unit vectors.
- 9) Calculate the u unit vectors
- 10) Calculate the theta's: joint angles

FINDINGTHETA

- 1) Converting the N and V vectors from Spherical to cartesian for calculations
- 2) Inverse kinematics
- 3) Calculate the side a unit vectors
- 4) Calculate the middle a unit vector
- 5) Multiply the a unit vectors by their geometric design length
- 6) Find the Cartesian coordinates of the corners and centerpoints of every unit cell
- 7) vert_cn: Initialize a matrix for the centerpoints
- 8) Calculate Cartesian coordinates by multiplying every v_vectors by their corresponding geometric design length.
- 9) Change all vectors back to unit vectors.
- 10) Calculate the u unit vectors
- 11) Calculate the theta's: joint angles

REALFREE

```
close all;
clear all;

global k1;
global k2;
global k_alpha;
global k_beta;
global k_lambda;
global h_scaling;
global a;
global v_vectors;
global n_norm;
global phi_ni;
global theta_ni;
global theta_vi;
global theta_tab;
global ncells;
global xcell;
global dist;
global xcellvar;
global shift_xcell;

k1=500;
k2=500;
dist=.4905;

ncells=[1,9];
xcell=[0, 0, 0;
      dist, 0, 0;
      dist*2, 0, 0;
      dist*3, 0, 0;
      dist*4, 0, -.1;
      dist*5, 0, 0;
      dist*6, 0, 0;
      dist*7, 0, 0;
      dist*8, 0, .2];

k_alpha=(58.054*pi)/180;
k_beta=(22*pi)/180;
h_scaling=1/(cos((k_beta)/2));
```

```

%%%%%%%%creating the 'on/' 'off' matrix for what value canchange%%%%%%%%
xcellvar=zeros(size(xcell));

```

```

counter31=0;
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1;
        if xcell(counter31,3)==0 %if xcell=0 note: '0.0000001' is not '0'
            xcellvar(counter31,1)=1; %then put these 1 or 0 into xcellvar
            xcellvar(counter31,2)=1; %1=free
            xcellvar(counter31,3)=1;
        else
            xcellvar(counter31,1)=1;
            xcellvar(counter31,2)=1;
            xcellvar(counter31,3)=0; %0=fixed
        end
    end
end
xcellvar;

```

```

%%%%%%%%additional constraints %%%%%%%%%%
%
% LT%    upper Lt 0 0 0 0 upperRT
% S%      0 0 0 0 0 0
% I%      0 0 0 0 0 0 ^
% D%      1stPT 0 0 0 0 LowerRT | Y
% E
% -----Bottom row-----
% -----X->-----> X

```

```

xcellvar(1,:)= [0 0 0];
% nXm size of the xcell_bdpt %constraint the first pt with all '0' b/c 1st pt is constraint

```

```

%%%%%%%%Constraining the various coordinate at the corners of the matrix %%%%

```

```

xcellvar(ncells(2)*ncells(1)-ncells(2)+1,3)=0; %constraint the upper LT Z coodinate
xcellvar(ncells(2)*ncells(1)-ncells(2)+1,1)=0; %constraint the upper LT x coodinate
xcellvar(ncells(2),3)=0; %constraint the lower RT Z coodinate
xcellvar(ncells(2),2)=0; %constraint the lower RT 1 coodinate

```

```

%if ncells(1)==1
% xcellvar(:,2)=0;
%end

```

```

% if ncells(2)==1
%   xcellvar(:,1)=0;
%end

xcellvar(ncells(1)*ncells(2),3)=0; %constraint the upper RT Z coodinate

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%moving xcell to orgin%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

flag_shift=0;
shift_xcell=xcell(1,3);
if xcell(1,3)~=0;
    flag_shift=1;
    xcell(:,3)=xcell(:,3)-xcell(1,3);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[Lx,Ly]=link_length(ncells,xcell);
%function that finds the length of link: this will show what the intial guess gives

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%% Flat non-deformed Cartesian C -> Spherical C %%%%%%%%%

%%%%%%%% initial guess %%%%%%%%%

[phi_n,theta_n,theta_v]=cc_convert_sc(xcell,xcellvar);%interpolation function

phi_ni=phi_n;
theta_ni=theta_n;
theta_vn=theta_v;

x0=[phi_n; %should be condensed
    theta_n;
    theta_v];

x=x0;

[xcell]=sc_graph(x,0); %function that graphs

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%bounds for options%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%lower bound for searching for guess

lower_b=[];
for it=1:3*ncells(1)*ncells(2)+1;
    lower_bi=-60*pi/180; %phi_ni's

    lower_b=[lower_b;lower_bi];
end

for it=1:3*ncells(1)*ncells(2)+1;
    lower_bi=-60*pi/180; %theta_ni
    lower_b=[lower_b;lower_bi];
end

for it=1:3*ncells(1)*ncells(2)+1;
    lower_bi=theta_vi(it)-60*pi/180; %theta_vi
    lower_b=[lower_b;lower_bi];
end

%upper bound for searching for guess
upper_b=[];
for it=1:3*ncells(1)*ncells(2)+1;
    upper_bi=60*pi/180; %phi_ni's
    upper_b=[upper_b;upper_bi];
end

for it=1:3*ncells(1)*ncells(2)+1;
    upper_bi=60*pi/180; %theta_ni
    upper_b=[upper_b;upper_bi];
end

for it=1:3*ncells(1)*ncells(2)+1;
    upper_bi=theta_vi(it)+60*pi/180; %theta_vi
    upper_b=[upper_b;upper_bi];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

options=optimset('Display','iter','MaxIter',1e5, 'MaxFunEvals',1e12,'TolFun',0.001,
'TolX',0.001);

tic;
x = fmincon('minimizePenergy',x0,[],[],[],[],lower_b, upper_b,[],options);

```



```

toc;
sec_elapsed_fmincon=toc

xcell

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[xcell]=sc_graph(x,flag_shift); %function that graphs
xcell

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Results from Fmincon is condensed, uncondense the variables%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

cnt=0;
phi_n=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;

    if mod(it,4)==0 & it>4 %MOD Modulus (signed remainder after division).
        phi_n3=phi_n(it-6);
        phi_n=[phi_n;phi_n3];
        cnt=cnt-1;% reseting counter to previous cnt becaused skipped a number
    else
        phi_n2=x(cnt);
        phi_n=[phi_n;phi_n2];
    end
end

theta_n=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;

    if mod(it,4)==0 & it>4 %MOD Modulus (signed remainder after division).
        theta_n3=theta_n(it-6);
        theta_n=[theta_n;theta_n3];
        cnt=cnt-1; % reseting counter to previous cnt becaused skipped a number
    else
        theta_n2=x(cnt);
        theta_n=[theta_n; theta_n2];
    end
end

theta_v=[];
for it=1:4*ncells(1)*ncells(2)

```

```

    cnt=cnt+1;
    if mod(it,4)==0 & it>4 %MOD    Modulus (signed remainder after division).
        theta_v3=theta_v(it-6)+pi;
        theta_v=[theta_v;theta_v3];
        cnt=cnt-1; % resetting counter to previous cnt becaused skipped a number
    else
        theta_v2=x(cnt);
        theta_v=[theta_v;theta_v2];
    end
end
end

```

```

[theta_tab, vert_centpt] = findingtheta(phi_n, theta_n, theta_v);

```

```

theta_tab

```

SC_GRAPH

```
function [xcell]=sc_graph(x,flag_shift)
%graphing the spherical values by uncompressing and changing to cartesians

global k1;
global k2;
global k_alpha;
global k_beta;
global k_lambda;
global h_scaling;
global phi;
global a;
global v_vectors;
global n_norm;
global phi_ni;
global theta_ni;
global theta_vi;
global theta_tab;
global ncells;
global xcell;
global dist;
global xcellvar;
global shift_xcell;

%%%% %%%%%%%%%%uncompression/undensing.....%%%%%%%%%
%%%%%%%%% HAS the repeats where all the n's that are parellel will be listed as well
%%%%%%%%% At the begining all Phi_ni=Phi_n,theta_ni=theta_n, and theta_vi=theta_v
%%%%%%%%% THE Phi_ni, etc were uses as initial guesses, while the Phi_n, etc.
%%%%%%%%% were kept as the orginal inputs and deformation
%%%%%%%%% In this section replace the results X values from 'fmincon' into
%%%%%%%%% the Phi_n,etc. but do not replace the constrained values.

cnt=0;
phi_n=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;

    if mod(it,4)==0 & it>4  %MOD   Modulus (signed remainder after division).
        phi_n3=phi_n(it-6);
        phi_n=[phi_n;phi_n3];
        cnt=cnt-1;% resetting counter to previous cnt because skipped a number
```

```

else
    phi_n2=x(cnt);
    phi_n=[phi_n;phi_n2];

end
end

theta_n=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;

    if mod(it,4)==0 & it>4 %MOD (signed remainder after division).
        theta_n3=theta_n(it-6);
        theta_n=[theta_n;theta_n3];
        cnt=cnt-1; % resetting counter to previous cnt because skipped a number
    else
        theta_n2=x(cnt);
        theta_n=[theta_n; theta_n2];
    end
end

theta_v=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;
    if mod(it,4)==0 & it>4 %MOD (signed remainder after division).
        theta_v3=theta_v(it-6)+pi;
        theta_v=[theta_v;theta_v3];
        cnt=cnt-1; % resetting counter to previous cnt because skipped a number
    else
        theta_v2=x(cnt);
        theta_v=[theta_v;theta_v2];
    end
end

%%%%%%%%%%
%%%%%%%%%%
%phi_n=[x(1); phi_ni(2); x(2); phi_ni(4)];
%theta_n=[x(3); theta_ni(2); x(4); theta_ni(4)];
%theta_v=[x(5); theta_vi(2); x(6); theta_vi(4)];

flagplot=1; % for plotting graphs in image3d

[theta_tab, vert_centpt] = image3d(phi_n, theta_n, theta_v,flagplot,flag_shift);
% Kinematic program that produces the thetas fo minmizations
if flag_shift==1
    vert_centpt(:,3)=vert_centpt(:,3)+shift_xcell;

```

```

end
xcell=vert_centpt;
[Lx,Ly]=link_length(ncells,xcell);
%function that finds the length of link: this will show what the I.G gives
graph_ctpt(ncells,xcell);% a function that graphs the center pt
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

MINIMIZINGPENERGY

```
function [pot_energy] = minimizePenergy(x)

global k1;
global k2;
global k_alpha;
global k_beta;
global k_lambda;
global h_scaling;
global phi;
global a;
global v_vectors;
global n_norm;
global phi_ni;
global theta_ni;
global theta_vi;
global theta_tab;
global ncells;
global xcell;
global dist;
global xcellvar;

%%%%%% spherical: n01=[ phi, theta] because r is always 1
%%%%%% theta= rotating latitude
%%%%%% phi= rotatin longitude

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Creating relations between vectors%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%This code sets any normal that should be parallel is parallel. %%%%%%%%%%%

%example: n2= n8 because both vectors are on the linking links
cnt=0;
phi_n=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;

    if mod(it,4)==0 & it>4 %MOD Modulus (signed remainder after division).
        phi_n3=phi_n(it-6);
        phi_n=[phi_n;phi_n3];
        cnt=cnt-1;% reseting counter to previous cnt becaused skipped a number
    else
        phi_n2=x(cnt);
        phi_n=[phi_n;phi_n2];
    end
end
```

```

    end
end

%theta_n=theta_ni;
theta_n=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;
    if mod(it,4)==0 & it>4 %MOD Modulus (signed remainder after division).
        theta_n3=theta_n(it-6);
        theta_n=[theta_n;theta_n3];
        cnt=cnt-1; % reseting counter to previous cnt becaused skipped a number
    else
        theta_n2=x(cnt);
        theta_n=[theta_n; theta_n2];
    end
end

theta_v=[];
for it=1:4*ncells(1)*ncells(2)
    cnt=cnt+1;

    if mod(it,4)==0 & it>4 %MOD Modulus (signed remainder after division).
        theta_v3=theta_v(it-6)+pi;
        theta_v=[theta_v;theta_v3];
        cnt=cnt-1; % reseting counter to previous cnt becaused skipped a number
    else
        theta_v2=x(cnt);
        theta_v=[theta_v;theta_v2];
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

flagplot=0; % 0 for not plotting graphs; 1=ploting
[theta_tab, vert_centpt] = image3d(phi_n, theta_n, theta_v,flagplot,0);
    % Kinematic functions that produces the thetas fo minmizations

theta_tab;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%separate calucations %%%%%%%%%

pot_energy=0;

for it2=1:ncells(1)*ncells(2)

```

```

for it=1:3:3*4; %for every sections(4sections total) there are three angles

    pot_energy1=1/2 *k1*(115.1005*pi/180-theta_tab(it,it2) )^2 ...
        +1/2 *k2*(82.6897*pi/180-theta_tab(it+1,it2))^2 ...
        +1/2 *k1*(115.1005*pi/180-theta_tab(it+2,it2))^2;

    pot_energy=pot_energy+pot_energy1;
end
end

%%%%%%%%%%creating adding constraints on fixed value%%%%%%%%%%

counter31=0;% interiopr cell conunter
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1; %%% interiopr cell conunter Start at the 1st interior cell
        if xcellvar(counter31,3)==0 %i0= constrained
            pot_energy=pot_energy+...
                100000*(vert_centpt(counter31,3)-xcell(counter31,3))^2;
        end
    end
end

counter31=0;% interiopr cell conunter
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1; %%% interiopr cell conunter Start at the 1st interior cell

        if ncells(1) ==1 %
            pot_energy=pot_energy+...
                100000*(vert_centpt(counter31,2)-xcell(counter31,2))^2;
        end
    end
end

counter31=0;% interiopr cell conunter
for it30= 1: ncells(1);% % rows
    for it31= 1:ncells(2)% % column
        counter31=counter31+1; %%% interiopr cell conunter Start at the 1st interior cell

        if ncells(2) ==1 %
            pot_energy=pot_energy+...
                100000*(vert_centpt(counter31,1)-xcell(counter31,1))^2;
        end
    end
end

```



```
    end  
  end  
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
pot_energy;  
close
```

IMAGE3D

```
function [theta_tab, vert_centpt] = image3d(phi_n, theta_n, theta_v, flagplot, flag_shift);
% Kinematic
%program that produces the thetas fo minmizations

global k1;
global k2;
global k_alpha;
global k_beta;
global k_lambda;
global h_scaling;
global phi;
global a;
global v_vectors;
global n_norm;
global phi_ni;
global theta_ni;
global theta_vi;
global theta_tab;
global ncells;
global xcell;
global dist;
global xcellvar;
global shift_xcell;

%%%%%%%% spherical: n01=[ phi, theta] because r is always 1
%%%%%%%% theta= rotating latitude
%%%%%%%% phi= rotatin longitude
%%%%%%%% There is a relationship of the phi for v vectors and the n vector locations

%%%%%%%%%converting the N and V vectors back to cartesian/rectangular for
calculations%%%%%%%%
n_norm=[];
%%converting back to rectangular for calculations
for it=1:4*ncells(1)*ncells(2)
    n_norm(it,:)=[sin(phi_n(it))*cos(theta_n(it)), sin(phi_n(it))*sin(theta_n(it)),
cos(phi_n(it))];
end

v_vectors=[];
%converting back to rectangular for calculations
for it= 1:4*ncells(1)*ncells(2)
    v_vectors(it,:)=[cos(phi_n(it))*cos(theta_n(it))*cos(theta_v(it))...
```

```

        - sin(theta_n(it))*sin(theta_v(it)), ...
        cos(phi_n(it))*sin(theta_n(it))*cos(theta_v(it))...
        + cos(theta_n(it))*sin(theta_v(it)), ...
        -sin(phi_n(it))*cos(theta_v(it))];
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%Calculating angles%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
a=[]; %a=side vector of top small connecting triangles

for it21=1:4*ncells(1)*ncells(2)
    a_val1=([v_vectors(it21,:)]-tan(k_beta/2)*(cross(n_norm(it21,:),...
        v_vectors(it21,:)))/(h_scaling); %vp1 +w_rot1
    a=[a;a_val1];
    a_val2=([v_vectors(it21,:)]+tan(k_beta/2)*(cross(n_norm(it21,:),...
        v_vectors(it21,:)))/(h_scaling); %vp1 +w_rot2
    a=[a;a_val2];
end

a; %%%% a8 %a1 %a2 %a3.....
%a=example for 1 unit cell:[a8; a1; a2; a3; a4; a5; a6; a7]
%rearrange the a matrix to move the first a value from each unit cell...
%%become the last value for each unit cell
a2=[]; %a1 %a2 %a3.....%a8

for it23= 1:8*ncells(1)*ncells(2)-7
    a1=[a(it23+1:it23+7,:);a(it23,:)];
    a2=[a2;a1];
end
a=a2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

m=[];
;
count4=0;
%for changing the vector 1 and vector2 from the 'a' matrix for 'For Loop it6'
%finding a2z=az_mid1 by searching through trial and error: in 'For Loop it5'
L1= 73/256; %%%length of the side of intermediate joints
L5=83/256; %%%length of the middle vector of intermediate joints
for it6=1:4*ncells(1)*ncells(2) % the total numbers of a_mid for all cells
    count4=count4+1;
    vect1=a(count4,:);
    count4=count4+1;
    vect2=a(count4,:);

```

```

p_vect=0.25*vect2 - 0.25*vect1;
    %%% p_vect= a vector that connects a(i) and a(i+1).
q_vect=0.25*vect1 + p_vect/2;
    %%%q_vect= vector taht connects the centerpt of unit cell to mid of p_vect.
L4= sqrt(L1^2 - (norm(p_vect/2))^2);
    %% length that connects Q_vector to the bottom of the triangle
angle_triangle= acos( -(L4^2)+(norm(q_vect))^2 +L5^2)/(2*norm(q_vect)*L5));
L6= norm(q_vect)*tan(angle_triangle);
    %%L6= perpendicular from end of q_vector to L5
n_avects= cross(vect1,vect2); % normal vector to 2 crossing a vectors
n_avectu= n_avects/(norm(n_avects)); % the unit vector of the a vector normal
m_vect= -L6*n_avectu+q_vect;
m_vectu= m_vect/(norm(m_vect));
m=[m;m_vectu];
end

ap=a*0.25; %scaler of a vectors
a_mid= m*(83/256) ; %scaler of m_mid vectors

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%turning the C.C of V and N into the C.C. of the centerpts for eachunit cell%%
%pat_vert=[0,0,0];
%vert_cn=[0,0,0];

vert_cn=zeros(ncells(1)*ncells(2),3); %starting off the vertices of the centerpts for each
unit cell

% calculates all the vertrices pts
cnt=0;
pat_vert=[];
cnt2=0;
for it2= 1:ncells(1)*ncells(2) % going through each unit cell
    pat_vert=[pat_vert; vert_cn(it2,:)];
    %first intializes the first vertices which is the centerpt for each unit cell
    for it=1:4
        cnt=cnt+1;
        cnt2=cnt2+1;
        pat_vert1=[ap(cnt,:); % adds in the vertices for the scaler "a vectors" and "middle
vectors"
        a_mid(cnt2,:)];
        pat_vert=[pat_vert; pat_vert1]; %stacking
        cnt=cnt+1;
        pat_vert2=ap(cnt,:);
        pat_vert=[pat_vert; pat_vert2];
    end
end

```

```

    end
end
pat_vert;
%translating groups of 13points all at the same time to the scalar postion
%for each unit cell
cnt=0;
cnt2=13; %starting at the 13th vertices which is on the 2nd cell
for it=2:ncells(1)*ncells(2)
    for it2=1:13 %for every cell there is 13 vertice points
        cnt2=cnt2+1; %conuter to count for each vertices
        pat_vert(cnt2,:)=pat_vert(cnt2,:)+pat_vert(13*cnt+1,:)+v_vectors(2+cnt*4,:)*dist;
    end
    cnt=cnt+1;
end

pat_vert;

cfaces=[];
vert_centpt=[];

cnt=0;
for it=1:ncells(1)*ncells(2)
    for it2=1:11
        cfaces1=[13*cnt+it2+1,13*cnt+1,13*cnt+it2+2];

        cfaces=[cfaces; cfaces1];
    end
    vert_centpt1=pat_vert(13*cnt+1,:); %placing all of the center points of each unit cell
    into one matrix
    vert_centpt=[vert_centpt;vert_centpt1];
    cfaces2=[13*cnt+2 ,13*cnt+1,13*cnt+13];
    cfaces=[cfaces;cfaces2];
    cnt=cnt+1;
end
vert_centpt;
cfaces;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if flagplot==1;
    color_vect=[]; %for RGB

    for it=1:ncells(1)*ncells(2)

```

```

        for it2=1:4
            color_vect1=[0,0,1; %blue face in RGB
                0,1,0; %green face in RGB
                1,0,0]; %red face in RGB
            color_vect=[color_vect; color_vect1];
        end
    end

    pat_vert1=pat_vert;
    if flag_shift==1
        pat_vert1(:,3)=pat_vert(:,3)+shift_xcell;
    end

    patch('Vertices',pat_vert1,'Faces',cfaces,'FaceVertexCData',color_vect,'FaceColor','flat')
    view(3);
    axis equal;
    pause(.2)
end

%%%%%%%%% change back to unit vectors%%%%%%%%%
m_unit=[];
for it =1:4*ncells(1)*ncells(2)
    m1= a_mid(it,:)/(norm(a_mid(it,:)));
    m_unit=[m_unit;m1];
end
m=m_unit;

a_unit=[];
for it =1:8*ncells(1)*ncells(2)
    a1= ap(it,:)/(norm(ap(it,:)));
    a_unit=[a_unit;a1];
end

ap=a_unit;
%%%%%%%%%

%findin the U vectors that are normal to the intermediate links using the 'a' and 'a_mid'
vectors
%a=example for 1 unit cell:[ a1; a2; a3; a4; a5; a6; a7; a8]
%a_mid= [a_mid1(between a1 and a2); a_mid2(between a3 and a4); a_mid3(between a5
and a6); a_mid4(between a7 and a8)]

u_norm=[];

```

```

count5=0;
for it7=1:4*ncells(1)*ncells(2)
    count5=count5+1;
    u_val1=( cross(ap(count5,:),a_mid(it7,:)) )/...
        ( norm (cross(ap(count5,:),a_mid(it7,:)) ));
    u_norm=[u_norm;u_val1];

    count5=count5+1;
    u_val2= ( cross(a_mid(it7,:), ap(count5,:)) )/...
        ( norm (cross(a_mid(it7,:), ap(count5,:)) ));
    u_norm=[u_norm;u_val2];
end
%example for 1 unit cell:[8 rows, 3 colns] [ u1; u2; u3; u4; u5; u6; u7; u8]

%%%%%explanation of finding theta%%%%%%%%%

% find theta1 (between main and intermediate), theta2 (between 2 intermediate), theta3
(between main and intermediate)
% g_n= [g_n1; g_n2; g_n3; g_n4];

% from paul's eqns:
% theta1=- acos(dot(u1,n1)+pi
% theta3= -acos(dot(u2,n2)+pi    modified from Paul's eqn: '+' => '-'.
% theta 2= -acos(u1,u2)+ pi

%%%%%%%%%

theta=[]; % for every 3 rows are the 3 thetas for 1 out of 4 section of the radial symmetric
unit cell.
%copying the beginning the 1st norm to the end for: 'For loop it8' because of circular
motion

for it=0:ncells(1)*ncells(2)-1
    count6=0;
    for it8= 1:4
        count6=count6+1;
        theta1= -abs(acos( dot(u_norm(count6+it*8,:), n_norm(it8+it*4,:))) + pi ;
        theta=[theta;theta1];

        theta2= -abs(acos( dot(u_norm(count6+it*8,:), u_norm((count6+1)+it*8,:))) +pi;
        theta=[theta;theta2];

        if it8==4
            theta3= -abs(acos( dot(u_norm((count6+1)+it*8,:), n_norm(1+it*4,:))) + pi ;
            theta=[theta;theta3];

```

```

        count6=count6+1;
    else
        theta3= -abs(acos( dot(u_norm((count6+1)+it*8,:), n_norm((it8+1)+it*4,:)))) + pi
    ;
        theta=[theta;theta3];
        count6=count6+1;
    end

end

end

theta_tab=[];
for it24=1:12:3*4*ncells(1)*ncells(2)
    theta_t=[theta(it24:it24+11)];
    theta_tab=[theta_tab,theta_t];
end

```

FINDING THETA

(WAS ALREADY SHOWN IN APPENDIX D AND WILL NOT BE REPEATED)

BIBLIOGRAPHY

1. Barr, A.H., "Global and Local Deformations of Solid Primitives". Proceedings of ACM SIGGRAPH, July 1984.
2. Bathe, K.J. "Finite Element Procedures in Engineering Analysis", Prentice-Hall, 1996.
3. Beitz, W. and Pahl, G. "Engineering Design: A Systematic Approach". 2nd Ed. Springer-Verlag London Limited. Great Britain 1996.
4. Bosscher, P. and Ebert-Uphoff, I., "A Novel Spherical Joint Mechanism". Accepted for presentation at IEEE Robotics and Automation Conference, 2003.
5. Bosscher, P. and Ebert-Uphoff, I., "Digital Clay: Architecture Designs for Shape-Generating Mechanisms". Accepted for presentation at IEEE Robotics and Automation Conference, 2003.
6. Bosscher, Paul "Digital Clay: Architecture Designs for Shape-Generating Mechanisms". Master Thesis. Georgia Institute of Technology. Atlanta, GA April 2003.
7. Capell, S., Green, S., Curless, B., Duchamp, T., and Popovic, Z. "A Multiresolution framework for Dynamic Deformations". Proceedings of ACM SIGGRAPH, 2002.
8. Chapra and Canale, "Numerical Methods for Engineers" 3rd Ed. WCB McGraw Hill Boston 1998.
9. Choi, K.S., Sun, H., Heng, P.A., and Cheng, J.C.Y. "A Scalable Force Propagation Approach for Web-Based Deformable Simulation of Soft Tissues". Proceedings of 7th International Conference on 3D Web Technology, ACM Press, 2002.
10. Coquillart, S., "Extended Freeform Deformation: A Sculpting Tool of 3D Geometric Modeling". Proceedings, ACM SIGGRAPH, August 1990.
11. DeBunne, G., Desbrun, M., Cani, M-P., and Barr, A.H., "Dynamic Real-Time Deformations using Space Time Adaptive Sampling". Proceedings ACM SIGGRAPH, 2001.
12. Diez, Jacob. "Design for Additive Fabrication: Building Miniature Robotic Mechanisms". Master Thesis Georgia Institute of Technology. Atlanta, GA March 2001.

13. Gill, Philip; Murray, Walter; and Wright, Margaret. "Practical Optimization". Academic Press Limited. San Diego, CA. 1981.
14. Ginsberg, Jerry. "Advanced Engineering Dynamics". 2nd Ed. Cambridge University Press. New York, NY 1998.
15. Goldfarb, M. and Speich, J., "A Well-Behaved Revolute Flexure Joint for Compliant Mechanism Design," ASME Journal of Mechanical Design, vol. 121, no. 3, pp. 424-429, 1999.
16. Gurocak, H., Parrish, B., Jayaram, S., Jayaram, U. "Design of a Haptic Device For Weight Sensation in Virtual Environments". Proceedings of ASME Computers and Information in Engineering Conference, paper #DETC2002/CIE-34387, Montreal, Canada, September 2002.
17. Immersion, www.immersion.com.
18. Iwata, H., Yano, H., Nakaizumi, F, and Kawamura, R., "Project FEELEX: Adding Haptic Surface to Graphics". Proceedings ACM SIGGRAPH, August 2001.
19. Jacobs, Paul F. Ph.D. "Rapid Prototyping & Manufacturing: Fundamentals of Stereolithography". Society of manufacturing Engineers. Dearborn, MI USA 1992.
20. Kota, Sridhar; Joo, Jinyong; Li, Zhe; Rodgers, Steven M.; and Sniegowski, Jeff. "Design of Complaint Mechanisms: Applicants to Mems". Analog Integrated Circuits and Signal Processing, 29, 7-15. Kluwer Academic Publishers. Netherland 2001.
21. Liu, He and Schubert, Daniel H., "Effects Of Nonlinear Geometric And Material Properties On The Seismic Response Of Fluid-Tank Systems", 2002 ANSYS 10th International Conference and Exhibition. Pittsburgh, PA 2002.
22. MathWorks, www.mathworks.com.
23. McDonnell, K.T. and Qin, H. "FEM-Based Subdivision Solids for Dynamic and Haptic Interaction". Proceedings of 6th Symposium on Solid Modeling and Applications, 2000.
24. MEMS and Nanotechnology Clearinghouse, www.memsnet.org.
25. Park, Jae-Hyoung. "Process Planning for Laser Chemical Vapor Deposition". Master Thesis. Georgia Tech. Atlanta, GA, April 2003.
26. Rapid Prototyping Manufacturing Institute, www.rpmi.marc.gatech.edu.

27. S. Dalmia, S.H. Lee, S. Bhattacharya, F. Ayazi, M. Swaminathan, "High-Q RF Passives on Organic Substrates Using a Low-Cost Low-Temperature Laminate Process". Proc. 2002 Symposium on Design, Test, Integration and Packaging of MEMS/MOEMS (DTIP 2002), Cannes, France, May 2002.
28. Sederberg, T.W. and Parry, S.R. "Freeform Deformation of Solid Geometric Models". Proceedings of ACM SIGGRAPH, August 1986.
29. SensAble Technologies, www.sensable.com.
30. Virtual Reality Technology, www.caip.rutgers.edu/vrlab.
31. W. J. Chen, and W. Lin. "A Miniature Gripper System for Optical Fiber Handling". SPIE Conference on Optomechatronic System III, 12 - 14 November 2002.
32. Yong Wang, Sue Ann Bidstrup, Guang Yuan, and Mark G. Allen "Printed-Wiring-Board Microfluidics for Thermal Management of Electronic Systems". ECS 201st meeting, 2002.
33. Zeiny, A. "Nonlinear Time-Dependent Seismic Response of Unanchored Liquid Storage Tanks". Phd. Dissertation in Civil Engineering. University of California. 2000-09-06